Logic Optimization and Code Generation for Embedded Control Applications

Yunjian Jiang
211 Cory Hall, Department of EECS
University of California, Berkeley CA 94720
wjiang@eecs.berkeley.edu

Robert K. Brayton
573 Cory Hall, Department of EECS
University of California, Berkeley CA 94720
brayton@eecs.berkeley.edu

ABSTRACT

We address software optimization for embedded control systems. The Esterel language is used as the front-end specification; Esterel compiler v6 is used to partition the control circuit and data path; the resulting intermediate representation of the design is a control-data network. This paper emphasizes the optimization of the control circuit portion and the code generation of the logic network. The new control-data network representation has four types of nodes: control, multiplexer, predicate and data expression; the control portion is a multi-valued logic network (MV-network). We use an effective multi-valued logic network optimization package called MVSIS for the control optimization. It includes algebraic methods to perform multi-valued algebraic division, factorization and decomposition and logic simplification methods based on observability don't cares. We have developed methods to evaluate a control-data network based on both an MDD and sum-of-products representation of the multi-valued logic functions. The MDD-based approach uses multi-valued intermediate variables and generates code according to the internal BDD structure. The SOP-based code is proportional to the number of cubes in the logic network. Preliminary results compare the two approaches and the optimization effectiveness.

Keywords

Esterel, Logic optimization, MDD, Code generation, Multiple-valued

1. INTRODUCTION

We are concerned with software implementation in a hardware/software codesign framework. We adopt a functional architecture design methodology, which has three phases: specification, optimization and implementation. The system functionality is specified using a high-level language like Esterel [1]; it is interpreted and translated into an intermediate representation. This is then optimized independent of any final implementation. Finally an architecture binding step decides the ultimate implementation strategy, whether in hardware or in software.

CODES 01 Copenhagen Denmark

There has been much research in the literature on software synthesis for Esterel. The Esterel compiler proposed by Edwards [2] operates on a concurrent control flow graph and translates an Esterel program directly into C. The scheme is limited to statically schedulable Esterel programs and there is no room for advanced logic optimizations. The circuit-based Esterel compiler v4 and v5 [3, 4], translates an Esterel program into logic gates and generates code from the gates. The code is slow because all logic computation is generated even if some is not needed because of idle cycles. There is improvement by applying lazy code evaluation [5] in this respect, but the computation of binary logic gates is not an efficient use of the CPU. The BDD-based code generation approach in the POLIS project [6, 7], generates code to exhaustively simulate the transition relation of the finite state machine. For large circuits it tends to generate very large size code.

We propose a code generation approach that uses multi-valued logic optimizations and exploits bit-level parallelism of the control structure. We use Esterel as the high-level specification language and the Esterel compiler v6 to partition the control circuit and data path; we translate the result into an intermediate control-data network representation. Algorithms to optimize the control portion of the network are applied. This includes: (a) grouping binary control variables into multi-value ones; (b) optimizing the control network itself by algebraic methods, don't care-based simplification, elimination and resubstitution; (c) generating and passing don't cares from the data path to the control logic. After the optimization, we generate efficient code to evaluate the multi-valued logic and code for the data computation.

In Section 2, we describe the intermediate control data network representation and the optimizations we apply at this level. Section 3 gives the two code generation methods for a multi-valued logic network evaluation. We present some preliminary results and discussion in Section 4 and conclude in Section 5.

2. CONTROL-DATA NETWORK OPTIMIZATION

A control-data network has control nodes and data nodes interconnected with wires, or variables. A *control* variable is a binary or a multi-valued variable that can take a finite number of values; a *data* variable is an integer. According to their input and output types, we categorize data nodes

Table 1: Node types in a control data network

node types	*operation*	*input*	*output*
control	logical	control	control
expression	arithmetic	data	data
multiplexer	assignment	control-data	data
predicate	predicate	data	control

into three types: data expression, multiplexer and predicate nodes. They are shown in the Table 1. The distinction of these node types clearly separates control variables from the data path, yet preserves the functional information of the original design. This creates new opportunities for logic optimizations.

Data oriented optimizations like common subexpression elimination, constant propagation and symbolic value numbering are left for traditional compilers. We focus on the optimization of the control structure using logic minimization techniques. Although data nodes are treated as black boxes for optimization algorithms, some internal information about them can be interpreted and used for more aggressive logic optimization.

2.1 Multi-Valued Logic Networks

Methods for optimizing a multi-valued logic network have been generalized from binary logic. These include algebraic decomposition [8], don't care based simplification [9], elimination and resubstitution.

A multi-valued combinational logic network, or *MV-network*, is a network of nodes. Each node represents a multi-valued function with a single multi-valued output and many multi-valued inputs. There is a directed edge from node i to node j, if the function at node j syntactically depends on the output variable at node i. The network has a set of primary inputs and a set of nodes which are designated as the outputs of the network. An intermediate format for representing such a network is BLIF-MV [10] used in the VIS system [11]. In general, a variable x_i is multi-valued and takes on values from the set $P_i = \{0, 1, ..., |P_i| - 1\}$. A **literal** of a MV-variable x is associated with a subset of values for that variable. For example, suppose x can take on 5 values $\{0, 1, 2, 3, 4\}$; then $x^{\{0,2\}}$ and $x^{\{1,2,4\}}$ are literals of x. The interpretation of $x^{\{0,2\}}$ is that it is a binary logic function which is 1 if x has either the value of 0 or 2, and 0 otherwise. Note that $x^{\{0,1,2,3,4\}} = 1$ since all five possible values appear in the literal. A **product term** or **cube** is a conjunction of literals and evaluates to 1 if each of the literals evaluates to 1. A **sum-of-products** (SOP) is the disjunction of a set of product terms. It evaluates to 1 if any of the products evaluates to 1.

Algebraic methods [8] include methods for finding common sub-expressions, semi-algebraic division, decomposing an MV-network, factoring an expression, and algebraic resubstitution. *Network simplification* [9] uses a generalization of compatible observability don't cares to minimize the logic of a node in the network. This performs Boolean resubstitution as well. *Elimination* merges a node into its fanouts. *Resubstitution* tries to substitute existing nodes into larger nodes in order to save cubes or literals. These algorithms are implemented in a software package called MVSIS, available at [12].

2.2 Data Path Optimization

Data nodes are treated as black boxes by control optimization methods; the functionality inside can not be changed. However the information contained in data nodes is useful in generating and passing logic don't cares for control optimizations. The following are two schemes that can be effective in optimization; they are yet to be implemented.

- *Data path don't cares* The output data variable of a multiplexer might be deselected by a second multiplexer. When this occurs, the variable controlling the first multiplexer is blocked by the one that controls the second. The set of minterms that produces the blocking value can be used as don't cares for the logic controlling the first multiplexer.
- *Don't care inheritance in data path* The output control signal of a predicate might be blocked by some don't care minterms. This don't care set can be passed down through the data path that drives the predicate and given to the control variables that control the multiplexers driving the predicate node.

3. CODE GENERATION

The code generation for data nodes is a direct transformation from the data expression into C code. We incorporate the technique of lazy code evaluation introduced by [5]. Data variables are not evaluated until they are requested by a fanout node. In this section, we mainly discuss two new methods for generating code to evaluate a multi-valued logic network. They are based on two representations of multi-valued logic functions.

3.1 Sum-of-products Code for Control Nodes

In this approach, the function for each MV-node is represented as a multi-valued sum-of-products. This is stored in a *table* structure, similar to that used in VIS [11] for representing BLIF-MV [10]. The table at a multi-valued node stores a set of cubes for each output value. The cubes are represented using positional notation. One of the output values is the default value and is omitted in the representation. The following example MV-node has 5 inputs with 18 total input values and 4 output values, value 3 being the default. We evaluate the function of a node on a minterm

x_1	x_2	x_3	x_4	x_5	z
100	1001	0011	110	1111	0
010	0110	1001	100	1000	0
001	0111	0100	111	0110	1
101	0101	0010	001	1101	2
101	1100	1111	101	0001	2

in the local fanin space by testing it against each cube in the SOP table. If the minterm is contained by a cube, the output is the value that owns this cube. The containment test is achieved with an *AND* instruction. This is based on the fact that $m \in C \Leftrightarrow m \cdot \overline{C} = \emptyset$. The cubes are stored in complement forms so that only one *AND* instruction per cube is needed for this test.

We compare two software implementation schemes. (a) Table-based approach stores the network structure and the cubes

for each node in the memory as static data variables. A separate subroutine `eval-network()` is generated to traverse the static data structure and evaluate the network. It is a fixed code for all networks and can be implemented rather efficiently. The data portion of the code has some overhead in keeping the network structure, but it scales well as the size of the network increases. (b) Flat-code approach embeds the cubes as constants inside the `eval-network()` procedure. There is no data structure overhead in storing the network and no control overhead in evaluating it.

The experiments show that the flat-code approach has comparable code size, but is much faster than the table-based approach. This is because in the table-based approach, (a) the `eval-network()` has to deal with the most general cases, and (b) a large amount of memory accesses for retrieving the cubes causes cache misses. The following is an example of the flat-code generated by MVSIS.

```
_PI0_START:   I[0] |= 1<<(PI[0]+0);
_PI1_START:   I[0] |= 1<<(PI[1]+4);
_NO_START:
              if((I[0] & 0xFF93) != 0)
              if((I[0] & 0xFF6C) != 0)
                  goto _NO_PART2;
              goto _NO_FOUND_1;
_NO_PART2:
              if((I[0] & 0xFFC6) != 0)
              if((I[0] & 0xFF39) != 0)
                  goto _NO_DEFAULT;
              goto _NO_FOUND_2;
_NO_DEFAULT:  value=0; goto _NO_END;
_NO_FOUND_1:  value=1; goto _NO_END;
_NO_FOUND_2:  value=2;
_NO_END:
```

The primary inputs are stored in vector `PI[]`. Each node has a bit vector storing the input minterm. The first several lines of code sets up the input vector `I[0]`; the code that follows performs the cube containment test for each output value except the default. Here we make the assumption that: (1) the total number of input values for each table does not exceed 32 so that they can fit in a single integer; (2) one of the output values is configured as the default and need not be involved in the evaluation (in this case value 0).

Since the evaluation process is sequential, it proceeds to the next cube if and only if the input minterm is not contained in any of the cubes that have been tested. Therefore, after a cube is tested, it can be used as don't cares to minimize the remaining functions to be tested. We call this *priority don't cares*.

Definition 1. Priority Don't Care Given a multi-valued function f with n values, $\{0, \ldots, n-1\}$, and a priority ordering of the values, $\{r_1, r_2, \ldots, r_n\}$, $0 \leq r_i \leq n$, the priority don't care for value function f_{r_i} is $PDC_{r_i} = \cup_{k=0}^{i-1} f_{r_k}$.

In the minimization process, the observability don't cares (ODC) and satisfiability don't cares (SDC) generated by node simplification are common to all value functions. Priority don't cares (PDC) are combined with ODC and SDC to minimize each individual value function. Obviously the last value in the priority ordering can always be minimized into an empty set.

The priority ordering problem for output values is difficult. We apply a heuristic ordering scheme. It assigns each value a cost, which is a weighted sum of the cube count C_i and the literal count L_i for the cover, i.e. $Cost_i = \alpha C_i + \beta L_i$. The priority is obtained by ordering the values from the least to highest cost. The value with the least cost has the highest priority and is evaluated first. The reasoning is (a) the generated code is directly related to the cube count of a cover. High priority values have less PDCs available for minimization. Thus values with less cubes should be given higher priority; (b) Larger functions with more minterms produce more PDCs for downstream functions. Thus values with less literals should be given higher priority. Experiments show that using PDCs the total cube count for evaluation can be reduced to as much as 50% for some examples.

3.2 MDD Code for Control Nodes

This approach uses an MDD representation for each node. We build the characteristic function of the output relation by introducing the output variable into the MDD. The output variable is treated as the last one in the MDD variable ordering. If the output is ordered before all inputs, the generated code would be a set of direct computation of the logic function without branches. According to [7], this results in larger and slower code. A mixed ordering approach with the output variables possibly appearing anywhere is not adopted in our approach for the same reason.

The MDD package [13] uses an internal BDD engine, and keeps a correspondence between a multi-valued variable and the internal binary variables that are used to encode it. From the internal BDD structure we generate an *if-then-else* type of code for each BDD node; a constant integer bit mask is generated to extract the desired binary bit of the intermediate multi-valued variable.

$\{0,1\}$	×	$\{0,1,2\}$	×	$\{0,1,2\}$	→	$\{0,1,2\}$
0		0		–	→	0
0		1		$\{0,1\}$	→	0
0		1		2	→	1
0		2		–	→	1
1		0		0	→	1
1		0		$\{1,2\}$	→	2
1		$\{1,2\}$		–	→	2

Above is an example of a multi-valued node with three inputs $\{u, v, w\}$ and one output $\{z\}$. Variable u is binary; variables v, w, z have three values. The table shows the sum-of-products representation of the output function for each value. The MDD representation for each value of z is shown in Figure 1(a). $u(2)$ denotes the second bit of variable u and $u(1)$ the first bit. In the BDD structures, solid edges denotes *then* branches; dashed edges denote *else* branches; black dots denote complement edges.

The characteristic function is built by adding the output variable at the bottom and combining the MDDs for each value. This is shown in Figure 1(b). This representation is similar to an ADD [14]. The size of the generated code is proportional to the number of BDD nodes in the MDD of the characteristic function. BDD dynamic variable reordering using sifting is performed after each characteristic function is built. The binary encoding variables are allowed to interleave as long as the output variable stays at the bottom. Experiments show that frequent dynamic reordering gives considerable improvement in code size. The code for

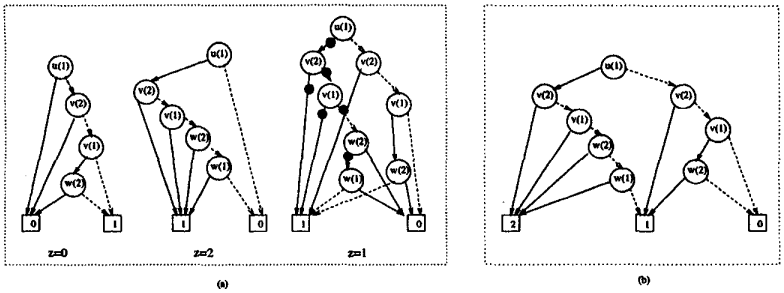

Figure 1: MDD construction of the characteristic function

the top several nodes of this example is shown below, where N0 denotes the index of this MV-node; L0 denotes the internal BDD node.

```
_N0_L0:   if (0x1 & u) { goto _N0_L1; }
          else         { goto _N0_L2; }
_N0_L1:   if (0x2 & v) { goto _N0_L3; }
          else         { goto _N0_L4; }
_N0_L2:   if (0x2 & v) { goto _N0_L5; }
          else         { goto _N0_L6; }
_N0_L3:   I[0] = 2;
          goto _N0_END;
```

Compared with other BDD-based code generation approaches [7], our approach combines multiple binary variables into a single multi-valued variable. It has the following advantages: (a) it consumes less memory for storing intermediate results; (b) it shares BDD nodes between multiple output values yielding less code size; (c) it updates multiple binary bits for the same MV-variable simultaneously, compared with updating each binary bit separately. Also, the multi-level multi-valued minimization and decomposition done in MVSIS serves as a method for clustering functionality into tables and hence into compatible MDDs.

4. EXPERIMENTS

The code generation methods are implemented in MVSIS, a software infrastructure for multi-valued logic optimization. We first compare the code size and performance of both MDD and SOP codes for pure multi-valued logic network examples. Most of the examples come from the multi-valued logic benchmark suite from Portland State University; some are hand-made examples. The characteristics of the examples are shown in Table 2. It shows, for each MV-network, the total number of primary inputs, followed in parenthesis by the ranges of their values, and the total number of primary outputs followed by the ranges of their values. Before generating code, the following optimization methods are applied iteratively until no further improvement in a cost function that estimates the final code size.

```
sweep; simplify; eliminate; merge
fast_extract; decompose; resubstitute
reset_default; full_simplify
```

Table 2: Multi-valued network examples

examples	PI	PO	*examples*	PI	PO
adder-mod4	3 (4)	1(4)	nursery	8 (3-5)	1 (5)
conv35	4 (2)	2(5)	pal3	6 (3)	1 (2)
decoder	3 (4)	2(8)	balance	4 (5)	1 (5)
plus8	2 (8)	1(15)	car	6 (3-4)	1 (4)
max	8 (8)	1(8)	mm3	5 (3)	1 (3)
matmul	8 (3)	4(3)	mm4	5 (4)	1 (4)
xor-mux	3 (4-6)	1(4)	mm5	5 (5)	1 (5)
ex2	5 (3)	2(2-3)	iris	4 (5-12)	1 (3)
ex5	7 (2)	2(2)	monks3te	6 (2-4)	1 (2)
employ1	9 (3-5)	1(4)	monks2tr	6 (2-4)	1 (2)
employ2	7 (3-5)	1(4)	monks3tr	6 (2-4)	1 (2)

In Table 3, the SIZE column shows object code size generated using `gcc -O3 -c` and measured by GNU utility `size`. The SPEED column shows simulation run time. We generate random input vectors and execute the evaluation code 1,000,000 times; the total run time is recorded using the GNU profiling tool `gprof`. The size of the code that generates test benches is subtracted from the total size.

In general these two methods are comparable in run time. The code size comparison varies for different examples. For small examples, MDD code tends to be smaller; for larger examples SOP code size is better. On average, the SOP code has smaller size. This gives one the option of trading off between speed and code size depending on different real-time applications.

We are not yet able to compare our results with other code generation approaches for Esterel since we currently do not have support for data path representations. We believe the techniques introduced in this paper, combined with new multi-valued logic optimization methods, will improve the quality of the code generated from Esterel and provide a useful method for trading run time and code size. In addition, these results are preliminary and not yet definitive in comparing the relative merits of the two methods.

5. CONCLUSIONS

We proposed a code optimization and generation scheme for embedded control applications. This includes an intermediate control data network representation, multi-valued logic optimization for control, and two methods for code generation from multi-valued logic networks. We showed some preliminary results for comparing the two code generation

Table 3: Comparison between SOP code and MDD code

	SIZE(byte)		SPEED(ms)	
examples	SOP	MDD	SOP	MDD
adder-mod4	238	264	110	160
conv35	262	940	90	190
decoder	290	280	220	100
plus8	618	456	150	130
max	1026	1740	410	170
matmul	1042	1360	330	320
xor-mux	210	200	120	110
ex2	198	320	80	90
ex5	174	136	60	80
employ1	342	608	130	110
employ2	318	808	140	100
nursery	390	540	200	140
pal3	154	260	130	50
balance	982	1180	200	90
car	378	728	120	110
mm3	150	256	50	110
mm4	422	672	130	140
mm5	870	1856	200	60
iris	338	1500	70	70
monks3te	58	100	80	20
monks2tr	442	1240	170	130
monks3tr	382	1092	160	190
average	1	184%	1	93%

approaches. By using multi-valued variables, we explore the bit-level parallelism of the generated real-time code. We believe this is faster and smaller than the code generated from the encoded binary version of the same logic network.

This is ongoing research. We expect to conduct more extensive experiments using larger logic examples as well as Esterel benchmarks. The next step is to incorporate data representation into MVSIS and develop logic optimization algorithms to take advantage of data path don't cares, data path don't care inheritance and possibly arithmetic optimization.

Acknowledgement

We are grateful for the support of the SRC under contract 683.004 and the California Micro program and industrial sponsors, Fujitsu, Cadence, Motorola and Synopsys.

6. REFERENCES

[1] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 1992.

[2] S. Edwards, "Compiling esterel into sequential code," in *Proc. of the Design Automation Conf.*, June 2000.

[3] G. Berry, "Esterel on hardware," *Philosophical Transactions of the Royal Society of London. Series A*, 1992.

[4] G. Berry, *The constructive semantics of pure Esterel.* Book in preparation, 1996.

[5] F. Balarin and M. Chiodo, "Software synthesis for complex reactive embedded systems," in *Proc. of the Intl. Conf. on Computer Design*, Oct. 1999.

[6] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synthesis of software programs from CFSM specifications," in *Proc. of the Design Automation Conf.*, June 1995.

[7] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 834–49, June 1999.

[8] M. Gao and R. K. Brayton, "Semi-algebraic methods for multi-valued logic," in *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2000.

[9] Y. Jiang and R. K. Brayton, "Don't cares and multi-valued logic network minimization," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2000.

[10] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang, "BLIF-MV: An Interchange Format for Design Verification and Synthesis," Tech. Rep. UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[11] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *IEEE International Conference on Computer-Aided Verification*, 1996.

[12] R. K. Brayton and et al., "MVSIS." `http://www-cad.eecs.berkeley.edu/Respep/Research/mvsis/`.

[13] T. Kam and R. K. Brayton, "Multi-valued deisoin diagrams," Tech. Rep. UCB/ERL M90/125, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Dec. 1990.

[14] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 188–91, Nov. 1993.