Generalized Cofactoring for Logic Function Evaluation

Yunjian Jiang Slobodan Matic Robert K. Brayton
University of California, Berkeley
{wjiang, matic, brayton}@eecs.berkeley.edu

ABSTRACT

Logic evaluation of a Boolean function or relation is traditionally done by simulating its gate-level implementation, or creating a branching program using its Binary Decision Diagram (BDD) representation, or using a set of look-up tables. We propose a new approach called generalized cofactoring diagrams, which are a generalization of the above methods. Algorithms are given for finding the optimal cofactoring structure for free-ordered BDD's and generalized cube cofactoring under an average path length (APL) cost criterion. Experiments on multi-valued functions show superior results to previously known methods by an average of 30%. The framework has direct applications in logic simulation, software synthesis for embedded control applications, and functional decomposition in logic synthesis.

Categories and Subject Descriptors

J.6 [**Computer-aided design**]; B.6.3 [**Logic Design**]: Design Aids—*Simulation*; D.2 [**Software**]: Software Engineering

General Terms

Algorithms Design

Keywords

Logic simulation, code generation

1. INTRODUCTION

Given a Boolean function, one can evaluate its output value for an input minterm by (a) simulating its gate-level network implementation, either through event-driven simulation [9] or levelized compiled code [7], (b) using a branching program derived from its Binary Decision Diagram (BDD) representation [10, 1], or (c) using a set of memory look-up tables. However, how to generate the optimum code for logic function evaluation on a given computer architecture is still unsolved.

In this paper, we study the optimal functional evaluation problem for a multi-valued relation. We use multi-valued logic, because at an early design stage, there may not be a binary encoding yet; relations are involved, because they capture flexibility and enable a larger exploration space. We propose to use generalized cofactoring for logic evaluation. A Generalized Cofactoring Diagram (GCD) is a decision structure for searching for the output value of a given input minterm. At each decision node, one of the out-going edges is chosen, based on the result of a testing of the input minterm and a cofactoring function associated with that node. The tests proceed along a path in the decision structure until a leaf node is reached, where the output value is obtained. GCD's reduce to BDD's if single variables are used as the cofactoring functions.

Since the cofactoring functions are not restricted to single variables, the framework opens up a large space for optimization. This is not unrealistic because the evaluation of a cube can be done in software in one instruction. The evaluation of an arbitrary function up to certain size is also feasible on an Application Specific Instruction-set Processor (ASIP) that has reconfigurable functional units [3]. The cofactoring functions can be implemented as special instructions for speeding up the critical path of the evaluation.

We give algorithms to derive an optimal GCD based on different constraints on the cofactoring functions. This includes an exact algorithm to find the optimum GCD when the cofactoring functions are single variables (i.e. free BDD's), and an heuristic algorithm based on entropy reduction when the cofactoring functions are single cubes. We propose a novel technique that symbolically represents all possible cube candidates in a single BDD, and computes their costs in parallel at each cofactoring step.

Related work in the literature includes using BDDs for logic simulation, proposed in [10, 1]. Subsequent work extended the techniques to handle larger circuits and circuits with data-path modules [8], but the restriction of single variable testing still applies.

The Polis project uses *S-graph*'s to represent and synthesize software from extended finite state machines [2]. It is similar to BDDs, with the extension of assignments for data-path variables. Recently, Kim *el al* [6], proposed free-ordered BDDs versus globally ordered BDDs for software synthesis in the same framework. Up to 10% improvement in execution speed was reported. However, the heuristics proposed do not guarantee optimality.

We briefly describe our definition of generalized cofactoring in Section 2, and detail our proposed techniques in Section 3. It includes optimal free BDD ordering (3.1) and symbolic cube selection for generalized cube cofactoring (3.2). Results are given in Section 4; conclusions and future work follow.

2. GENERALIZED COFACTORING

Given a multi-valued relation, find the optimum cofactoring structure that minimizes a cost function related to the functional evaluation time of the structure[1].

DEFINITION 1. *A multi-valued relation $R(a,b,\ldots,O)$ is a relation $R: A \times B \times \cdots \times O \mapsto \{0,1\}$, where $a,b,\ldots$ are multi-valued variables in the input domain taking on values from the sets $A,B,\ldots$ respectively, and O is the output domain.*

A generalized cofactoring diagram is a DAG, with a single root node and a number of leaves, each corresponding to a subset of the output values. Each intermediate node is associated with a 2-tuple $< R, g >$, where R is the MV relation to be evaluated at this point, and g is a cofactoring function chosen for R. If we restrict g to be binary functions, then each node has two out-going edges,

[1]Multi-valued relations appear in a multi-level logic network when complete flexibility is used [11].

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

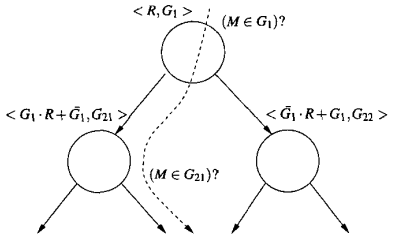


Figure 1: Generalized cofactoring for relations

representing the positive and negative cofactors: $(g \cdot R + \overline{g})$, $(\overline{g} \cdot R + g)$. A node becomes a leaf, if the relation R can be reduced to tautology, i.e. a common value can be assigned to all minterms in the care set without violating the relation.

Functional evaluation of an input minterm M starts from the top node and follows one of the paths towards the leaves. At each intermediate node, a test $(M \in g)$ decides which cofactoring branch to follow. The evaluation time of the cofactoring diagram can be measured by the average depth of the paths for all input minterms, used by Sasao, *el at* [13] for BDDs.

DEFINITION 2. *The Average Path Length (APL) of a GCD is the average number of branching nodes encountered during the functional evaluation for all its input minterms.*

Assuming the testing of each intermediate node requires one memory lookup, APL measures the average number of memory accesses for evaluating the output. Other factors like computer architecture and memory organization may also affect the evaluation time but not considered at this abstraction level. We also assume a uniform distribution of probability for all input minterms. The problem of considering bias input probabilities is beyond the scope of this paper.

3. OPTIMAL COFACTORING FOR FUNCTIONAL EVALUATION

This section describes our technique for deriving the optimal GCD for an MV function, with variable cofactoring and cube cofactoring. In either case, the MV function is represented with a multi-terminal BDD (MTBDD) or ADD, where multi-valued input variables are encoded with binary variables. Note that when MV functions are considered, there is exactly one terminal node for each output value, and no terminals for subsets.

3.1 Free BDD

By free BDDs we mean BDDs without global variable ordering, and each path can take a different order. This was studied by Kim, *el at*, [6] for software synthesis from EFSMs, but the results reported there are not optimal. Algorithm BPL (Best Path Length) below computes recursively the best ordering for each path of the BDD, with respect to the APL cost.

```
BPL (f) {
  if f is constant, return 0;
  if support_size(f) == 1, return 0;
  if hash_lookup(f,&pl), return pl;
  best_cost = Infinity;
  foreach input v in support_list(f) do {
      cost = 1 + ½BPL(f_v) + ½BPL(f_v̄);
      if cost < best_cost then best_cost = cost; }
  hash_insert(f,best_cost);
  return best_cost; }
```

THEOREM 1. *The BPL algorithms returns the optimum average path length for all possible free BDD orderings.*

At each step of the recursion, all possible orderings of the support variables are enumerated, and the best cost is returned. The best variable choice at each recursion step is stored to construct the final cofactoring structure at the end (not shown in the Figure). The complexity of this algorithm is upper bounded by (3^n), where n is the number of input variables in the support. This is the case when all possible cofactors are derived. Experiments show that MV functions with up to 16 binary input variables can be computed within seconds, more in Section 4.

3.2 Cube Cofactoring

If we restrict to multi-valued single cubes for the cofactoring function, the branching test can be carried out in a single *AND* instruction, assuming both the cofactoring cube C and the input M are represented in a positional notation:

$$m \in C \iff m \cdot \overline{C} = 0$$

Previously proposed MDD and SOP-based approaches [4] can be viewed as special cases of the generalized cube cofactoring: the MDD approach uses single literals as cofactoring functions and the SOP approach uses the cubes directly from a sum-of-product representation of the function.

In this section, we propose a novel symbolic method for finding the optimal cofactoring cube at each step, based on a cost function that predicts the cofactoring depth. For a GCD with arbitrary functions as the cofactoring functions, it can be achieved through collapsing a set of input variables into a single multi-valued variable, whose literals represent all possible functions of the selected inputs [5].

3.2.1 Cost Function for Cube Selection

Let $F : A, B, \ldots \mapsto O$ be the MV function to be evaluated, where $A = \{0, 1, \ldots, r_a\}$,$B = \{0, 1, \ldots, r_b\}$, etc, form its input domain, and $O = \{o_0, o_1, \ldots, o_{r_o}\}$ is its range [2]. An example:

	F		
b\a	0	1	2
0	1	2	1
1	2	0	0
2	3	1	2
3	0	3	3

$$F : \{0,1,2\} \times \{0,1,2,3\} \mapsto \{0,1,2,3\} \quad a \in \{0,1,2\} \quad b \in \{0,1,2,3\} \tag{1}$$

We need a cost function for measuring the complexity, in terms of output evaluation, of the care set of an MV function. Entropy, as an information measurement, became a natural choice: the cofactoring process is a process of information discovery, and the effort can be measured by the amount of information to be discovered. As we proceed along a path in the GCD, more information is obtained with respect to the input minterm. When we reach a terminal node, the output is determined and there is no information left (zero entropy). We use the output entropy for a function f:

$$H(R_f) = -\sum_{i \in outputs} p_i \cdot ln(p_i) = -\sum_{i \in outputs} \frac{m_i}{M_c} \cdot ln(\frac{m_i}{M_c}) \tag{2}$$

where p_i is the probability of the output taking value i, computed by dividing the number of minterms in i (m_i) by the total number of care set minterms (M_c). R_f is the relation for the care set of MV

[2]In the implementation, an MV function is represented as a relation using ADD's, and only the care set minterms are taken into consideration

function f. Given a cube C, we compute the entropies of both the positive and negative branches:

$$H(R_f|C) = p_c \cdot H(C \cdot R_f) + (1 - p_c) \cdot H(\overline{C} \cdot R_f) \quad (3)$$

where p_c is the probability of the cube C being true, computed by dividing the number of minterms in the intersection $C \cdot R_f$ with the total number of care set minterms in R_f. It takes into account of potential costs for both branches and their probabilities. In mathematical terms, this is the *conditional entropy* of R_f with respect to partition $\{C, \overline{C}\}$, which is always less than the original entropy: i.e.

$$H(R_f|C) \leq H(R_f)$$

and the difference of the two is called the *mutual information*. In reducing $H(R_f|C)$, we search for the cube partition that has the largest mutual information with the original relation:

$$max\{I(R_f, C) = H(R_f) - H(R_f|C)\}$$

The Example (1) above cofactored by cube $a^{\{1,2\}}b^{\{1,3\}}$ gives cost: $\frac{4}{12}(ln2) + \frac{8}{12}(\frac{2}{8}ln(8) + \frac{6}{8}ln(\frac{8}{3})) = 1.068$, which is a reduction from the original entropy: $ln(4) = 1.386$.

3.2.2 Solution Relations

We introduce new binary variables to symbolically represent all cube candidates in a positional notation. Let a *symbolic cube* be represented as

$$C_{sym} = (\alpha_0\alpha_1 \ldots \alpha_{r_a}\beta_0\beta_1 \ldots \beta_{r_b} \ldots)$$

where α_i means that the cube has value i in the a-literal, etc. For example, $\alpha_0\alpha_1\overline{\alpha_2}\beta_0\overline{\beta_1}\beta_2\overline{\beta_3}$ encodes cube $a^{\{0,1\}}b^{\{0,2\}}$. The total number of cubes encoded, except null cubes, is $(2^{r_a} - 1)(2^{r_b} - 1)\cdots$.

The logic function corresponding to each symbolic cube is called its *boolean cube* C_{bool}. In order to evaluate both the positive and negative cofactoring branches, we construct the cube relation C_{rel} for all cube candidates:

$$C_{rel} = C_{sym} \cdot z \cdot C_{bool} + C_{sym} \cdot \overline{z} \cdot \overline{C_{bool}}$$

where z is called the *phase* variable. $z = 1$ ($z = 0$) means the positive (negative) phase of the cube is used, respectively. The example cube above leads to the cube relation below:

$$\alpha_0\alpha_1\overline{\alpha_2}\beta_0\overline{\beta_1}\beta_2\overline{\beta_3}(za^{\{0,1\}}b^{\{0,2\}} + \overline{z}a^{\{2\}} + \overline{z}b^{\{1,3\}})$$

We build the Boolean disjunction of C_{rel} for all cubes and intersect this with the relation of the target MV function. This is called the *solution relation*.

$$R(a, b, \ldots) \cdot \sum_{i \in allcubes} C^i_{rel}(\alpha_0\alpha_1 \ldots \beta_0\beta_1 \ldots, z, a, b, \ldots) \quad (4)$$

The *solution relation* encodes all candidate cubes for both their positive and complement phases. It provides a convenient basis for computing the cofactoring cost of all cubes in parallel.

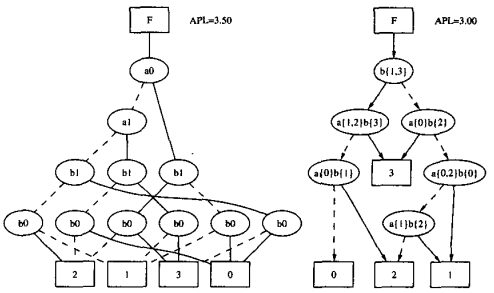
THEOREM 2. *The number of BDD nodes for the phase variable z, in the solution relation BDD is exactly the number of symbolic cubes, and each has exactly one parent node.*

Based on this theorem, the cost functions for the cubes need to be computed only at the z variable nodes of the BDD. Other nodes are traversed for passing information and cube selection. The negative side is that the solution relation BDD grows with the number of cubes generated. We address this in Sections 3.2.4.

We propose an algorithm *Select_Cube* to traverse the BDD structure of the *solution relation* bottom up, compute the cost functions incrementally, and construct the cube with the best cost (details in [5]). It assumes a variable ordering that groups the symbolic variables on the top, followed by the z phase variable, and then the boolean variable group comes last. Dynamic variable ordering within each group is allowed.

$$\{\alpha_0, \alpha_1, \ldots, \beta_0, \beta_1 \ldots\}, \{z\}, \{a, b, \ldots\}$$

The algorithm works in three phases. Phase I traverses the boolean variables at the bottom group. The number of minterms for each output value is computed. We store the information of each path in a data structure that contains the total number of care set minterms, the set of output values, and the minterm count for each value, for the sub-BDD. Phase II compute the cost functions of all cube candidates at the phase variable nodes (z). The costs from both children, representing C and $\overline{C}$, are weighted by their probabilities and then summed (Equation (3)). Finally, phase III traverses the symbolic variables in the top group. At each node, the costs from both children branches are compared; the path with the better cost is kept, and combined with the symbolic variable of the node itself. The algorithm returns a single cube (a path from top node to z), which has the best cost.



(a) MTBDD (b) GCD

Figure 2: Average Path Length (APL) comparison of the globally ordered MTBDD and the cube GCD computed with the entropy cost function (ENTR) for example *noname*

3.2.3 APL Look Ahead

Notice that the cost function in equation (3) is a local heuristic, which looks at one level cube cofactoring. We would like to get better prediction by considering the average path length of the whole sub-tree if subsequent cofactoration is carried out with this cube.

At each cofatoring step, we use the entropy cost to generate a set of candidate cubes. Then for each candidate we construct the full cofactoring structure for both its positive and negative branches, and compute the average path lengths for them. The cube with the best APL is selected. The sub-GCD construction uses the same *Select_Cube* algorithm described earlier for selecting the best cofactoring cube at each step. The procedure can be found in [5].

The comparison of the final cofactoring diagram for Example (1) is shown in Figure 2, where the cofactoring nodes for the MTBDD are the binary variables, a_0, a_1, b_0, b_1, used for encoding, and the ones in the GCD's are multi-valued cubes in positional notation.

3.2.4 Two Literal Cubes

Notice that the total number of cubes grows exponentially with the support size, and so does the size of the solution relation BDD. For a five 4-valued input function, there are $(2^4 - 1)^5 \simeq 759K$

	PI	IN	O	APL				SIZE (# nodes)				Computation time (sec)		
				BDD	FBDD	ENTR	LAHF	BDD	FBDD	ENTR	LAHF	FBDD	ENTR	LAHF
noname	2	12	4	3.50	3.50	3.00	2.67	11	11	6	6	0.00	0.02	0.06
adder	3	64	4	6.00	6.00	7.73	7.73	19	63	30	30	0.00	1.55	8.76
maxmin	3	125	5	5.21	5.21	5.15	4.71	48	68	32	30	0.08	33.07	167.68
xor_all	3	96	4	6.00	6.00	5.69	5.69	28	79	32	32	0.01	7.07	34.15
alg	4	625	5	5.04	2.62	1.70	1.70	74	57	8	8	0.52	4.95	33.61
balance	4	625	5	6.66	5.59	4.46	4.32	84	196	56	57	1.91	13.39	121.27
mm4	5	1024	4	5.77	4.77	4.23	3.98	69	83	29	29	0.15	2.89	15.73
mm5	5	3125	5	6.85	6.09	5.17	5.07	182	307	69	70	34.04	43.99	395.14
employ	7	18000	4	6.30	3.49	1.95	1.95	123	555	33	33	147.83	19.91	144.21
ex5	7	128	2	4.05	2.64	2.17	2.17	7	13	7	7	0.00	0.13	0.33
sort_b1	8	6561	3	5.40	5.40	2.23	2.23	23	510	8	8	5.28	1.11	3.38
sort_b2	8	6561	3	9.02	9.00	4.49	4.49	52	1792	38	38	26.43	76.17	76.47
sort_b3	8	6561	3	10.42	10.19	6.50	6.50	79	2814	92	92	68.92	76.34	75.99
average ratio				1.00	0.87	0.72	0.71	1.00	5.35	0.73	0.72	1.00	62.46	297.36

Table 1: Average Path Length (APL) and Generalized Cofactoring Diagram (GCD) size comparison on multi-valued functions. (The average ratio includes other examples that could not be shown due to space limit.)

cubes. It is expensive and not necessary to represent all these cubes. We apply heuristics such that for functions with large support sizes, we restrict to two-literal cubes, which have much better scalability. The same five 4-valued input function would generate $5 \times 2 \times (2^4 - 1)^2 = 2250$ cubes.

4. EXPERIMENTS

The *BPL* and *Selete_Cube* algorithms have been implemented using the CuDD package [14]. We compare the GCD derived from these approaches in Table 1. Benchmarks are two-level multi-valued functions obtained mostly from the Portland State University POLO suite [12]. The table only lists a small set and the complete list is availabe from [5]. The number of inputs and the sizes of the input domain are shown in columns `PI` and `IN`; their output domain sizes are shown in column `O`. Results from four different approaches are shown: BDD's with global variable ordering (BDD), optimum free-ordered BDD's (FBDD), GCDs with entropy cost (ENTR), and GCDs with APL look ahead cost (LAHF). They are compared with respect to the average path length (APL), the number of cofactoring nodes in the GCD structure, and finally the computation time used to derive these structures. Their ratios are computed for each benchmark example, and the average ratio of all 35 examples is reported in the last line. All experiments were performed on a dual-processor Sun OS 5.8 system, with 900MHz clock frequency and 2Gb memory.

The free-ordered BDD's have on average 13% better APL than globally ordered BDD's (after dynamic variable ordering to reduce the heap size). Although the size of the free-ordered BDD's is 5 times larger on average, it is still comparable on small and median examples. The size blows up on large examples since it is more difficult to find equivalent results in the hash table. The computation time is much higher than the globally ordered BDD's (whose computation time is negligible compared with the other approaches and not shown).

Comparing GCD's with BDD's, GCD's are consistently better in most examples, since they have a much larger solution space than free BDD's. The APL look ahead scheme is slightly better than just using entropies for a few examples (albeit the much heavier computation time), which confirms the quality of the entropy measure. Additional experiments on a large set (~ 50) of binary-input binary-output examples also show similar characteristics [5]. The APL look-ahead scheme starts to pay off as the sizes of the examples grow larger.

5. CONCLUSIONS

We presented a new framework for functional evaluation of multi-valued relations, which is a generalization of traditional logic simulation techniques. It has direct application in logic simulation, and software synthesis from state machines for embedded control applications.

Under this framework, we presented an algorithm to find the optimum free BDD ordering for fast evaluation, measured by the average path length of the cofactoring structure. For cube cofactoring, we proposed a symbolic representation of all cube candidates, and the *solution relation* that encodes all possible solutions for one cofactoring step. An algorithm is given to traverse solution relation and to find the optimal cube based on a cost function using functional entropies. Experimental results on binary and multi-valued functions justify the proposed approach. Future work includes devising efficient methods to compute cube costs incrementally and thus scalable to much larger examples.

Acknowledgement

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

6. REFERENCES

[1] P. Ashar and S. Malik. Fast functional simulation using branching program. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 408–412, Nov. 1995.

[2] 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. Comput.-Aided Design Integrated Circuits*, 18(6):834–49, June 1999.

[3] F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW RISC core. In *European Solid-State Circuits Conference*, Sept. 2001.

[4] Y. Jiang and R. K. Brayton. Software synthesis from synchronous specifications using logic simulation techniques. In *Proc. of the Design Automation Conf.*, June 2002.

[5] Y. Jiang, S. Matic, and R. K. Brayton. Generalized cofactoring for logic function evaluation. Technical report, University of California, Berkeley, 2003. http://www-cad.eecs.berkeley.edu/~wjiang.

[6] C. Kim, L. Lavagno, and A. Sangiovanni-Vincentelli. Free MDD-based software optimization techniques for embedded systems. In *Proc. of the Conf. on Design Automation & Test in Europe*, Mar. 2000.

[7] D. M. Lewis. A Hierarchical Compiled Code Event-Driven Logic Simulator. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 10(6):726–37, June 1991.

[8] Y. Luo, T. Wongsonegoro, and A. Aziz. Hybrid techniques for fast functional simulation. In *Proc. of the Design Automation Conf.*, pages 664–7, June 1998.

[9] P. M. Maurer. Event driven simulation without loops or conditionals. In *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2001.

[10] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 402–407, Nov. 1995.

[11] A. Mishchenko and R. K. Brayton. Simplification of non-deterministic multi-valued networks. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 557–62, Nov. 2002.

[12] Portland Logic Optimization group. Portland state university. `http://www.ee.pdx.edu/~polo`.

[13] T. Sasao, Y. Iguchi, and M. Matsuura. Comparison of decision diagrams for multiple-output logic functions. In *Proc. of the Intl. Workshop on Logic Synthesis*, Jun. 2002.

[14] F. Somenzi. CUDD: CU Decision Diagram Package. Technical report, University of Colorado, Boulder, 2001.