Generalized Cofactoring for Logic Function Evaluation

Yunjian Jiang Slobodan Matic Robert K. Brayton
Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720
{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. Experimental results on multi-valued functions are superior to previously known methods by 30%. The framework has direct application in logic simulation, software synthesis for embedded control applications, and functional decomposition in logic synthesis.

I. 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 [1] or levelized compiled code [2], (b) using a branching program derived from its Binary Decision Diagram (BDD) representation [3][4], 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.

This problem also appears in the context of software synthesis for embedded control applications. There, a design is described in a synchronous programming language and then compiled into a set of extended finite state machines (EFSM). These are mapped to a partition of hardware and software implementations depending on the system constraints [5][6]. The software implementation of an FSM is the code for its logic evaluation. Different from logic simulation on a high performance computer server, this code has tight constraints in real-time response and memory usage [7].

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.

This is an extended version of the same paper that will appear in DAC 2003, Anaheim, CA.

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 [8]. 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. Its relationship with logic decomposition is also discussed.

Related work in the literature includes using BDDs for logic simulation, proposed in [3], [4]. Subsequent work extended the techniques to handle larger circuits and circuits with data-path modules [9], 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 [10]. It is similar to BDDs, with the extension of assignments for data-path variables. Recently, Kim *el al* [11], 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.

Some of the ideas in this paper were inspired by recent advancements in Boolean decomposition using symbolic techniques [12] [13]. Generalized cofactoring can be viewed as a special type of Boolean decomposition with restrictions on the composition function (being a multiplexer), and the optimization criterion (average evaluation time rather than worse case delay or circuit area). However, the symbolic techniques we describe in this paper is transferable.

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 (III-A) and symbolic cube selection for generalized cube cofactoring (III-B), and generalized Boolean function cofactoring (III-C). Results are given in Section IV; conclusions and future work follow.

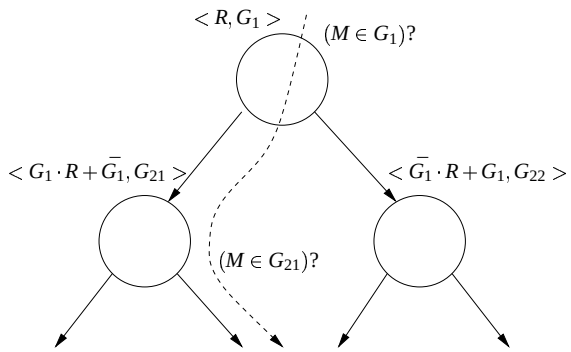


Fig. 1. Generalized cofactoring for relations

II. 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 outgoing edges, 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* [15] 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.

III. 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 [16], 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.

[1]Multi-valued relations appear in a multi-level logic network when complete flexibility is used [14].

A. 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*, [11] 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 + 1/2 BPL(f_v) + 1/2 BPL(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 average path lengthes from both the positive and negative branches are divided in half, because each branch has 0.5 probability of being taken. The best variable choice at each recursion step is stored and returned 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 IV.

B. Cube Cofactoring

It is computationally prohibitive to allow an arbitrary function as the cofactoring function g, and the the branching test $(M \in g)$ may not be efficiently implemented in software (on a general purpose embedded processor). If we restrict to multi-valued single cubes, 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 [17] can be viewed as special cases of 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 (ideas inspired by [12]) for finding the optimal cofactoring cube at each

step, based on a cost function that predicts the cofactoring depth. We then provide a slightly improved look-ahead cost function for better prediction, and two-literal restrictions for dealing with large scale functions.

B.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

$b \backslash a$	0	1	2
0	1	2	1
1	2	0	0
2	3	1	2
3	0	3	3

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

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 [18], 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:

$$\begin{aligned} H(R_f) &= -\sum_{i \in outputs} p_i \cdot ln(p_i) \\ &= -\sum_{i \in outputs} \frac{m_i}{M_c} \cdot ln(\frac{m_i}{M_c}) \end{aligned}$$

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

$$H(R_f|C) = p_c \cdot H(C \cdot R_f) + (1 - p_c) \cdot H(\overline{C} \cdot R_f) \tag{2}$$

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: $ln(4) = 1.386$.

[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

```
Select_Cube (ADD dd) {
  if is_constant(dd) {      //return symb struct with constant
      return symb(constant(dd)); }
  if hash_lookup(dd, &symb) {  //previous computed ADD node
      return symb; }
  symT = Select_Cube(THEN(dd));
  symE = Select_Cube(ELSE(dd));
  if is_bool_var(dd) then {  //accumulate minterms for each value
      symO.min = combine_minterms(symT,symE);
      symO.val = combine_values (symT,symE);
      return symO; }
  if is_z_var(dd) then {  //compute weighted entropy
      symO.cost = entropy(symT)*symT.min/care_minterms
                + entropy(symE)*symE.min/care_minterms
      return symO; }
  //now deal with symbolic variables
  if symT.cost < symE.cost then {
      symO.cube = attach_literal(symT.cube, lit(dd));
  } else {
      symO.cube = attach_literal(symE.cube, lit_not(dd)); }
  hash_insert(dd,symO);
  return symO;   }
```

Fig. 2. Algorithm Select_Cube: finding best symbolic cube

B.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:

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

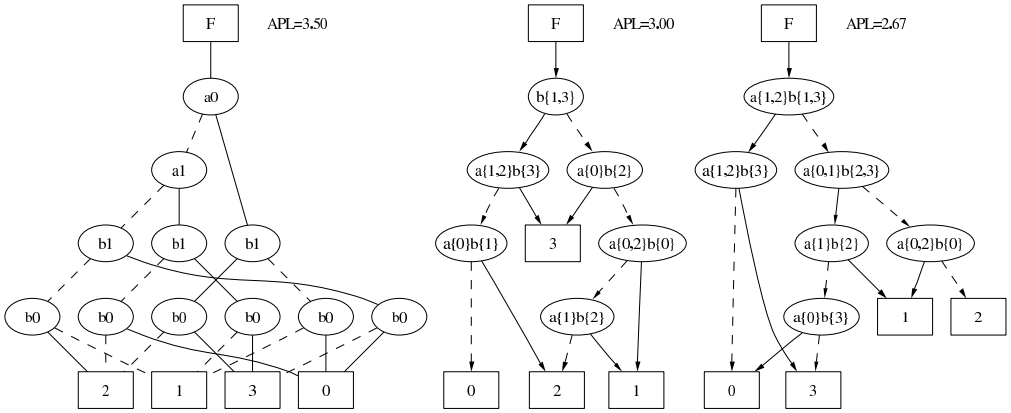
where z is called the *phase* variable. $z = 1$ ($z = 0$) means the positive (negative) phase of the cube is used, respectively. The example cube above leads to the cube relation below:

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

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*.

$$R(a, b, \ldots) \cdot \sum_{i \in allcubes} C^i_{rel}(\alpha_0\alpha_1 \ldots \beta_0\beta_1 \ldots, z, a, b, \ldots) \tag{3}$$

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.



(a) MTBDD with global ordering (b) GCD with entropy cost (c) GCD with APL look-ahead cost

Fig. 3. Average Path Length (APL) comparison of MTBDD, GCD with entropy (ENTR) and APL-look-ahead (LAHF) cost functions on example *noname*

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 III-B.4.

The algorithm *Select_Cube* traverses the BDD structure of the *solution relation* bottom up, computes the cost functions incrementally, and constructs the cube with the best cost (Figure 2). 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.

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

A hash table is maintained so that each node is guaranteed to be visited only once. After checking for terminal cases and the hash table, the algorithm is recursively called on both the positive branch (THEN()) and the negative branch (ELSE()) of the current node. During the boolean variable traversal phase at the bottom group, the number of minterms for each output value is computed. It is stored in a data structure (*symb*) 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. Output values are stored in a bit-vector, so that a bit-wise OR gives the union of the values from sub-branches (routine *combine_values*). At each phase variable node z, the costs from both children, representing C and $\overline{C}$, are weighted by their probabilities and then summed (Equation (2)). Finally, during the symbolic variable phase, the costs of both children branches are compared; the path with better cost is kept, and combined with the symbolic variable of the node itself. Routine *attach_literal* takes a literal and attaches it to the path (cube) obtained from one of the children branches. The algorithm finally returns a single cube, (a path from top node to z) which has the best cost.

B.3 APL Look Ahead

Notice that the cost function in equation (2) 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 sub-GCD construction uses the same *Select_Cube* algorithm described earlier for selecting the best cofactoring cube at each step. This procedure is described in the pseudo code below.

Look_Ahead_Cost (C, R_f) {
  if is_tautology(R_f) **return** 0;
  T = Solution_Relation(C, R_f);
  E = Solution_Relation($\overline{C}$, R_f);
  C_P = Select_Cube(T);
  C_N = Select_Cube(E);
  cost = 1+Prob(C,R_f)·Look_Ahead_Cost(C_P,$C \cdot R_f$)
    Prob($\overline{C}$,R_f)·Look_Ahead_Cost(C_N,$\overline{C} \cdot R_f$)
  return *cost*; }

Solution_Relation refers to the symbolic relation described previously, and *Prob*(C,R_f) computes the probability of cube C being true in the care set of relation R_f.

The comparison of the final cofactoring diagram for example (1) is shown in Figure 3, 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. In this example the look ahead scheme gives the best average path length. (Free MTBDD gives the same result as ordered MTBDD for this case.)

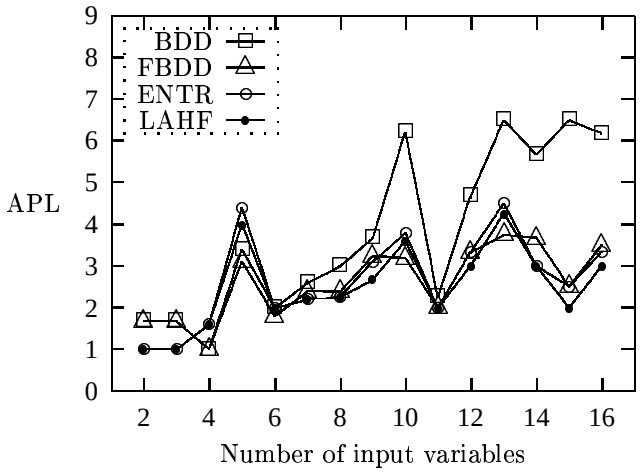


Fig. 4. Average Path Length (APL) comparison of BDD (BDD), Free-orderd BDD (FBDD), GCD with entropy as the cost function (ENTR) and the APL-look-ahead scheme (LAHF) on Boolean functions from MCNC benchmarks

B.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 $(2^4-1)^5 \simeq 759K$ 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.

C. Generic Function Cofactoring

Generic function cofactoring can be done through collapsing a set of input variables into a single multi-valued variable, whose literals represent all possible functions of the original inputs. Let $x_0, x_1, \ldots, x_{n-1}$ be a set of binary variables. Create multi-valued variable $\mu \in \{0, 1, \ldots, 2^n\}$, where each output value corresponds to one input minterm in the $x_0, x_1, \ldots, x_{n-1}$ domain.

Theorem 3: The literals of MV variable μ encode all possible Boolean functions on $x_0, x_1, \ldots, x_{n-1}$.

This is similar to a decoder with n inputs and 2^n outputs, where each output decodes one of the input patterns. Techniques in [19] can be applied to choose bound set variables for a decomposition; these are merged into a single multi-valued variable and then techniques presented in Section III-B applies for finding the optimal cofactoring function.

Recent work on combinational logic synthesis [12], [13] points to new directions of Boolean function decomposition using symbolic techniques, which are more powerful than traditional algebraic methods, and now become computationally affordable. The generalized cofactoring framework presented here is a special type of logic decomposition, with the restriction that a multiplexer is used for the composition function at each step. The goal of the optimum average path length is also different from logic synthesis in a synchronous circuit setting, where the worst case delay is concerned.

IV. Experiments

The *BPL* and *Selete_Cube* algorithms have been implemented using the CuDD package [20]. We compare the GCD derived from these approaches in Table I. Benchmarks are two-level multi-valued functions obtained mostly from the Portland State University POLO suite [21]. 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 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.

Figure 4 compares the same four approaches on a different set (~ 50) of binary-input binary-output examples, with support size ranging from 2-16. These are two-level functions extracted from collapsing multi-level networks from the MCNC benchmarks. The APL numbers are computed for all benchmark examples, and those with the same number of input variables are averaged. The average APL for each support level is drawn for each approach. The results again justify the generalized cube cofactoring approach compared with BDD's and free-ordered BDD's (with a few exceptions). Also the APL look-ahead scheme starts to pay off as the sizes of the examples grow larger.

The comparison above assumes that the average path length is the dominant factor in deciding logic function evaluation time in software. Other factors like software implementation and computer architectures are assumed to be equal for all approaches compared herein.

V. 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, functional equivalence checking, as well as software synthesis from state machines for embedded control applica-

				APL				SIZE (# nodes)				Computation time (sec)		
	PI	IN	O	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
conv0	4	16	5	4.00	4.00	3.50	3.50	12	15	8	8	0.00	0.02	0.09
conv1	4	16	5	4.00	3.00	2.62	2.62	10	8	5	5	0.00	0.02	0.02
mat_c12	4	81	3	5.25	5.25	5.39	5.39	32	48	21	21	0.01	0.78	5.84
mat_c11	4	81	3	5.25	5.25	5.39	5.39	32	48	21	21	0.02	1.23	1.67
mat_c22	4	81	3	5.25	5.25	5.39	5.39	32	48	21	21	0.20	1.43	1.69
mat_c21	4	81	3	5.25	5.25	5.39	5.39	32	48	21	21	0.10	1.74	1.80
ex2	5	243	3	4.44	3.96	2.92	2.89	30	35	10	10	0.05	4.25	18.43
ex3	5	243	3	4.44	3.96	2.92	2.89	30	35	10	10	0.05	4.24	18.41
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
ex4	6	64	2	3.28	3.28	2.34	2.34	6	12	6	6	0.00	0.04	0.15
monks	6	432	2	6.93	5.97	6.95	5.99	73	145	79	73	0.24	1.32	12.36
monks3	6	432	2	5.79	4.58	4.19	4.19	71	90	69	69	0.19	0.90	6.74
monksl	6	432	2	6.51	5.40	5.01	5.01	71	106	69	69	0.19	0.94	9.24
pal2	6	64	2	4.75	3.50	3.06	3.06	21	21	6	6	0.00	0.05	0.22
pal3	6	729	2	6.39	3.79	3.32	3.32	91	78	38	38	0.45	0.85	5.64
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
sort_b4	8	6561	3	10.28	9.52	7.13	7.13	96	2560	168	168	106.57	89.03	87.93
sort_b5	8	6561	3	9.29	7.91	5.86	5.86	96	1470	160	160	108.03	88.76	85.79
sort_b6	8	6561	3	7.56	5.99	4.18	4.18	79	544	98	98	64.18	80.24	77.77
sort_b7	8	6561	3	5.25	4.00	2.69	2.69	52	126	38	38	22.66	74.22	73.65
sort_b8	8	6561	3	2.66	2.00	1.33	1.33	23	16	8	8	3.47	69.85	71.27
average ratio				1.00	0.87	0.72	0.71	1.00	5.35	0.75	0.74	1.00	61.56	294.67

TABLE I

AVERAGE PATH LENGTH (APL) AND GENERALIZED COFACTORING DIAGRAM (GCD) SIZE COMPARISON ON MULTI-VALUED FUNCTIONS

tions. It is a special type of logic decomposition with different constraints and optimization criteria.

Under this framework, we presented an algorithm for finding the optimum free BDD ordering for fast evaluation, measured by the average path length of the cofactoring structure. In searching for the optimum cube cofactoring, we proposed a symbolic representation of all possible cube candidates, and the *solution relation* that encodes all possible solutions for one cofactoring step. An algorithm is given to traverse the BDD structure 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.

In future research, we will further explore the generalized function cofactoring for ASIP platforms. We will apply the generalized cube cofactoring technique in embedded software synthesis, where further constraints in code size may be enforced.

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.

REFERENCES

[1] P. M. Maurer, "Event driven simulation without loops or conditionals," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2000.

[2] D. M. Lewis, "A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 726–37, June 1991.

[3] 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*, pp. 402–407, Nov. 1995.

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

[5] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B.Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

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

[7] S. Edwards, L. Lavagno, E. A. Lee, and A. L. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proc. of the IEEE*, 1997.

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

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

[10] 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.

[11] 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.

[12] A. Mishchenko and R. K. Brayton, "A boolean paradigm in multi-valued

logic synthesis," in *Proc. of the Intl. Workshop on Logic Synthesis*, pp. 173–7, Jun. 2002.

[13] V. N. Kravets and K. A. Sakallah, "Resynthesis of multi-level circuits under tight constraints using symbolic optimization," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 687–93, Nov. 2002.

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

[15] 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.

[16] 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.

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

[18] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–56, July, October 1948.

[19] R. K. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions," in *Proc. of the Intl. Symposium on Circuits and Systems*, pp. 49–54, May 1982.

[20] F. Somenzi, "CUDD: CU Decision Diagram Package," tech. rep., University of Colorado, Boulder, 2001.

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