Sibling-Substitution-Based BDD Minimization Using Don't Cares

Youpyo Hong, *Member, IEEE*, Peter A. Beerel, *Member, IEEE*, Jerry R. Burch, and Kenneth L. McMillan

***Abstract*—In many computer-aided design tools, binary decision diagrams (BDD's) are used to represent Boolean functions. To increase the efficiency and capability of these tools, many algorithms have been developed to reduce the size of the BDD's. This paper presents heuristic algorithms to minimize the size of the BDD's representing incompletely specified functions by intelligently assigning don't cares to binary values. Experimental results show that new algorithms yield significantly smaller BDD's compared with existing algorithms yet still require manageable run-times. These algorithms are particularly useful for synthesis application where the structure of the hardware/software is derived from the BDD representation of the function to implement because the minimization quality is more critical than the minimization speed in these applications.**

***Index Terms*—Binary decision diagrams (BDD's), incompletely specified functions, sibling-substitution.**

I. Introduction

THE EFFICIENT representation and manipulation of Boolean functions is critical to many computer-aided design applications including logic synthesis, formal verification, and testing. Binary decision diagrams (BDD's) [1] have proven to be an efficient means of representing and manipulating many commonly used Boolean functions. For BDD-based tools, the size of the BDD's can determine their run-time efficiency, the problem size that they can handle and/or the quality of the circuits or software they synthesize. This paper focuses on BDD-based synthesis applications in which the quality of minimization is more critical than the minimization run-time.

The size of BDD's are heavily affected by the variable orders and many techniques have been developed to find BDD variable orderings that lead to compact BDD's [2]–[5]. Among many types of BDD's, reduced ordered BDD's (ROBDD's) are most widely used ones in practice. For a given variable ordering, the ROBDD representation of a completely specified function is unique. For an incompletely specified function, however, many ROBDD's can be used to represent the function, each associated with a different assignment of don't cares (DC's) to binary values. This paper assumes the variable ordering is fixed and addresses the problem of finding an assignment of DC's that yields a small ROBDD representation.

There are many synthesis tools in which circuits are directly derived from their BDD functional representation. For example, hazard-free multilevel logic based on multiplexor-based circuits can be directly derived from BDD's [6], [7]. In addition, T. Karoubalis *et al.* showed that differential cascode voltage switch (DCVS) logic circuits, which have many potential advantages such as performance and high layout density, can be optimally synthesized from BDD's due the canonicity of BDD's [8]. Moreover, Lavagno *et al.* presented a BDD-based timed Shannon circuits synthesis tool in which reducing the BDD size can lead to lower power consumption [9].

In the technology mapping area, multiplexor-based field programmable gate array (FPGA) mapping can be directly performed on the BDD's that represent the logic functions [10], [11]. Chang *et al.* applied DC-based BDD minimization in their FPGA mapping framework to reduce the size of the BDD's representing subject graphs, yielding more area-efficient circuits [11].

The application of BDD's can also be found in software synthesis area. Chiodo *et al.* [12] use a BDD as an intermediate representation to generate a software program because of the close connection between the BDD representation of a function and the structure of the software program they synthesize. The size of the software is determined by the BDD size, which means that the size of the BDD is critical to reduce software size [13], [14].

For incompletely specified functions, many BDD's can be used to represent the function; each associated with a different assignment of binary values to DC's. Finding the assignment that leads to the smallest BDD is known to be NP-complete [15] and exact techniques [16], [17] are typically too computationally expensive. Therefore, heuristic algorithms have been developed to address this *BDD minimization problem* [11], [18]–[20]. These heuristics try to maximize the instances of *node sharing* or *sibling-substitution* [19] during the minimization process. BDD nodes become *shared* if the reassignment of DC's makes their associated functions identical. *Sibling-substitution* is a special case of node sharing where a child of a BDD node u is replaced by the other child of u. Sibling-substitution leads to fewer nodes because a parent and its two children are replaced by the child when the two children are made identical.

Restrict and *constrain* (also known as *generalized-cofactor*) [18], [21], [22] are well known BDD minimization algorithms

Manuscript received July 16, 1998; revised May 26, 1999. The work performed at the University of Southern California was supported, in part, by a gift from Intel and an NSF CAREER Award MIP-2502386. The work performed at Dongguk University was supported by the research/publication encouragement program of Dongguk University. This paper was recommended by Associate Editor E. Macii.

Y. Hong is with the Electronic Engineering Department, Dongguk University, Seoul, Korea (e-mail: yhong@cakra.dongguk.ac.kr).

P. A. Beerel is with the Electrical Engineering-Systems Department, University of Southern California, Los Angeles, CA 90089 USA (e-mail: pabeerel@usc.edu).

J. R. Burch is with Cadence Berkeley Laboratories, Berkeley, CA 94704-1103 USA (e-mail: jrb@cadence.com).

K. L. McMillan is with Cadence Berkeley Laboratories, Berkeley, CA 94704-1103 USA (e-mail: mcmillan@cadence.com).

Publisher Item Identifier S 0278-0070(00)01371-3.

based on sibling-substitution and often achieve significant size reduction. Interestingly, however, these algorithms can yield a BDD far from being optimal and in fact larger than the original BDD.

Chang *et al.* [11] proposed a heuristic that makes multiple sub-BDD's shared by assigning DC's to binary values while traversing the BDD from top to bottom level by level. The reduction potential of their method is large, but its high computational complexity prohibits its application to large BDD's.

Shiple *et al.* [19] proposed a framework to relate sibling-substitution-based heuristics and Chang's heuristic, and explored their variants. Their experimental results suggest that sibling-substitution-based heuristics, specifically restrict and its variants typically outperform others in terms of both run-time and resulting BDD size.

Recently, Drechsler *et al.* proposed evolutionary algorithm-based BDD minimization algorithms that can handle multiple-output functions [20]. However, the minimization quality in this approach is very sensitive to some user-defined parameters [17].

Note that all the existing heuristics may produce larger results and a common way to avoid using a larger BDD is to compare the original BDD with the "minimized" BDD and use the smaller one. We refer to this approach as *thresholding*. The potential for the size increase, however, suggests that these methods may not produce BDD's as small as those produced by algorithms that *inherently* guarantee that no sub-BDD becomes larger.

This paper describes *safe* BDD minimization heuristics, i.e., they guarantee the resulting BDD is not larger than the original BDD inherently. These algorithms are based on sibling-substitution because sibling-substitution itself is very powerful and efficient. The key idea of *safe* minimization heuristics is to perform sibling-substitution only on the nodes that we can guarantee will not cause an overall increase in BDD size. These techniques can lead to better minimization results by preventing sibling-substitutions that can cause overall size growth while allowing sibling-substitutions elsewhere. Our heuristics can also be applied to minimize multiple BDD's safely.

Our experimentations on ISCAS and MCNC benchmarks using various types of DC's demonstrate that our new heuristics outperform existing sibling-substitution based heuristics significantly in minimization quality. Another strength of our heuristics is their low computational complexity which allows them to be able to minimize large BDD's that cannot be handled by competitive existing heuristics.

The organization of the paper is as follows. After defining the problem and presenting relevant notations in Section II, we present three new heuristics in Section III. We report our experimental results in Section IV and present conclusions in Section V.

II. Preliminaries

An m-input n-output Boolean function is a mapping $B^m \rightarrow B^n$. A Boolean function f can also be described as the set of all input points, i.e., minterms, for which the function f evaluates to 1; this set is referred as the "on-set" of f. Similarly, the "off-set" of f is the set of all input points for which the function f evaluates to 0. If we do not care if the function evaluates to 0 or 1 for a set of input points, i.e., the function is defined over a subset of B^m, the function is *incompletely specified*, and such input points are called the "don't care-set." Therefore, the domain of any single-output Boolean function ff can be partitioned into three subsets, $ff_{\rm on}$ (the on-set), $ff_{\rm off}$ (the off-set), and $ff_{\rm dc}$ (the don't care-set). A completely specified function has $ff_{\rm dc}$ empty, while an incompletely specified function has a nonempty don't care set. Any two of these sets uniquely describes an incompletely specified function.

An incompletely specified function ff can be represented by a pair of completely specified functions $[f, c]$ for which $f_{\rm on} \supseteq ff_{\rm on}$, $f_{\rm off} \supseteq ff_{\rm off}$, and $c = ff_{\rm on} \cup ff_{\rm off}$. For a given incompletely specified function ff, there are many such functions f, each referred to as a *cover* of ff [19], [23], representing different partitions of $ff_{\rm dc}$ into $f_{\rm on}$ and $f_{\rm off}$.

A BDD represents a function as a graph [1]. A BDD can have two types of nodes; leaf nodes and nonleaf nodes. The leaf nodes are either 0 or 1, representing the Boolean functions 0 and 1, respectively. Each nonleaf node u has two outgoing edges; a *then-edge* and an *else-edge*. Each edge is connected to a *child* node of u and u is the *parent* of the child nodes. The two child nodes are *siblings* of each other. Each nonleaf node F is associated with a Boolean variable x. The child of F reached via the then-edge is called the positive cofactor of F with respect to x and is denoted by F_x; the other child is called the negative cofactor of F with respect to x and is denoted by $F_{\overline{x}}$. The cofactor of F with respect to a cube[1] is the successive cofactoring of F with respect to all the literals in the cube. Each node F represents a Boolean function f. The size of F, denoted $|F|$, is the number of nodes in the BDD rooted at F.

An OBDD is a BDD with the constraint that the input variables are ordered and input variables appear in ascending order in every path from root to leaves. An ROBDD is an OBDD where there is only one node representing a distinct function. Bryant [1] proved that ROBDD's are canonical, i.e., the ROBDD representation of a completely specified function is unique under a fixed variable ordering. ROBDD's have practically proven most useful because of their canonicity and compactness. Therefore, we focus on ROBDD's and we refer to ROBDD's simply as BDD's in this paper.

An incompletely specified function can be represented by a BDD pair $[F, C]$ describing a pair of completely specified functions $[f, c]$, where f is a cover of the incompletely specified function and c denotes the care-function. Among all covers of ff, there must be at least one cover f' whose BDD F' is smallest in size. Unfortunately, finding a smallest F' is NP-complete [15], so we consider heuristic approaches. Given $[F, C]$, finding an F' that is hopefully close to minimal in size is called *BDD minimization using don't cares*. We call F the original BDD and F' the *minimized* BDD.

III. Safe BDD Minimization Based on Sibling-Substitution

The main differences among sibling-substitution based BDD minimization techniques lie in the criteria on which they per-

[1] A cube is a set of literals and represents the function obtained by their product [24].

form sibling-substitution. The simplest criterion is based on the observation that a function ff can be covered by any function if ff corresponds to a DC function. We refer to this condition as *DC-substitutability* which is formally defined as follows.

Definition 1—DC-Substitutability: ff is DC-substitutable if $ff_{\text{on}} \cup ff_{\text{off}} = \emptyset$.

The most widely used heuristic, restrict, is based on DC-substitutability. Restrict recursively traverses F and C concurrently in depth-first order. In each recursive call, a new pair of F and C nodes is visited and restrict checks if either of the positive or negative cofactor of the F node is DC-substitutable with respect to their corresponding C nodes. Whenever a cofactor is found DC-substitutable, sibling-substitution is performed to the parent (removing the parent and one child), and the result is built recursively by continuing traversal only to the cofactors that are not substituted.

Note that restrict, while always reducing the size of the target sub-BDD, can increase the size of a BDD that contains the target sub-BDD, as illustrated in Fig. 1.[2] Consider the node c in Fig. 1(a) which can be reached from the root by two different *paths* and has two different associated care subsets, represented by node d and leaf-1, respectively in C depicted in Fig. 1(b). The different pairs of node c and associated care subsets will be analyzed in different recursive calls of restrict. Consider first the recursive call in which the negative cofactor of node c (with respect to the variable c) in F, node d, is analyzed with the corresponding care node d in C. Because the care node is not leaf-0, the algorithm recurs to node d and analyzes its positive and negative cofactors. Since d's negative cofactor corresponds to a DC (leaf-0 in C), sibling-substitution is applied to d (replacing d with its positive cofactor leaf-0). Notice that this results in a smaller sub-BDD rooted at c. However, when node c is analyzed with the care node leaf-1 in a subsequent recursive call, the sub-BDD rooted at c (including node d) cannot be reduced at all because the entire sub-BDD corresponds to a care subfunction. Consequently, node c becomes unshared or *split*, and this *node-splitting* leads to the overall size increase illustrated in Fig. 1(c).

We can formally describe node-splitting using a relation R from edges in F to nodes in C defined as follows.

Definition 2: Given two BDD's F and C, the then-edge (else-edge) e of a node u in F is *related* to a node v in C iff there exist cubes p and $p' = p \cup \{x\} (p' = p \cup \{\overline{x}\})$ such that u denotes f_p, u_x denotes $f_{p'}$, and v denotes $c_{p'}$, where x is the variable associated with u and f_p is the cofactor of f with respect to p. We denote a set of nodes in C that an edge e in F is related to by $R(e)$.

Intuitively, the set $R(e)$ describes the set of care nodes that are visited during the recursive calls of restrict obtained by recurring through the edge e. More precisely, the set $R(e)$ consists of those care nodes analyzed in recursive calls: 1) in which the target of e is analyzed and 2) that are called by recursive calls in which the source of e is analyzed. For example, consider F and C shown in Fig. 1. For the cubes $\overline{a}$ and $\overline{a}\overline{b}$, $F_{\overline{a}}$ is denoted by the node b_1 in F, $F_{\overline{a}\overline{b}}$ by c in F, and $C_{\overline{a}\overline{b}}$ by d in C. According to Definition 2, the else-edge of b_1 in F is related to d in C. Intuitively, this means that in the only recursive call through the else-edge of b_1, the corresponding care node is d.

[2]In this paper, we label each node in a BBD by the variable that the node is associated with. When more than two nodes are associated with the same variable, numeric subscripts are used to distinguish the nodes. The then-edge (else-edge) is represented by a solid line(broken line).

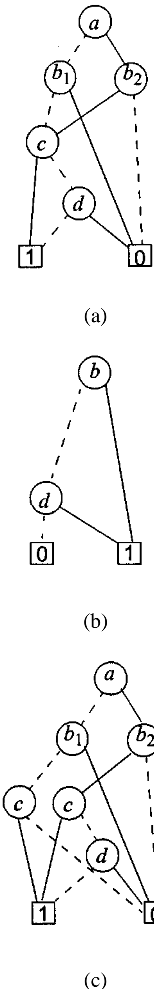


Fig. 1. Restrict example: (a) F, (b) C, and (c) F'.

As another example, consider the cubes $\overline{a}\overline{b}$ and ab. Both $F_{\overline{a}\overline{b}}$ and F_{ab} are represented by c in F. $F_{\overline{a}\overline{b}c}$ and F_{abc} are represented by leaf-1 in F, and $C_{\overline{a}\overline{b}c}$ and C_{abc} are represented by d and leaf-1, respectively, in C. Consequently, the then-edge of c in F is related to both d and leaf-1 in C. This makes sense because the then-edge of c is recurred through twice, once with the corresponding care node d and a second time with the care node 1.

The related nodes of all edges of F are shown in Fig. 2.

Notice that restrict applies sibling-substitution to u if the related nodes of its outgoing edge e includes leaf-0, i.e., leaf-0 $\in R(e)$. Node-splitting may occur if $R(e)$ also includes a non-DC because the original node (or a modified version of it) is needed in the result in such case. Consequently, an originally shared node, such as u, can become unshared by the minimization process, leading to overall BDD size growth.

We note that DC-substitutability is a sufficient but not necessary condition for a node to be able to substitute its sibling as pointed out by Shiple *et al.*in [19]. They developed variants of restrict using relaxed criteria that allow more sibling substi-

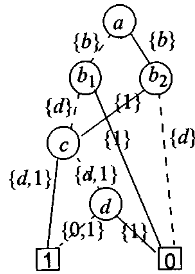


Fig. 2. An example of the relation R (0 and 1 mean leaf-0 and leaf-1, respectively).

```
bdd compact (bdd f, bdd c) {
    if (c == 0) return (0);
    mark-edges(f, c);
    result = build-result(f);
    return(result);
}
```

Fig. 3. Top level pseudocode for the proposed heuristics.

tutions. However, like restrict, many of these sibling substitutions cause node-splitting and consequently often cause BDD size growth.

We define general-substitutability as follows.

Definition 3—General-Substitutability[3]: ff is substitutable by gg if gg is a cover of ff.

We develop safe minimization heuristics by performing only the sibling-substitution that we can guarantee will not cause an overall BDD size increase. In other words we further constrain the condition to perform sibling-substitution even if substitutability holds.

Our algorithms basically consist of two phases. In the first phase, called *mark-edges*, the original BDD is preprocessed to conservatively identify nodes for which applying sibling-substitution does not increase overall BDD size. In the second phase, called *build-result*, sibling-substitution is selectively applied to the nodes identified in the first phase. In the trivial case when C is zero BDD, the zero BDD is returned as a minimization result without calling mark-edges. The top-level pseudocode for our algorithms is presented in Fig. 3.

We present three different compaction algorithms. The first, called *basic compaction*, performs a subset of the sibling-substitutions that we can conservatively guarantee do not lead to node-splitting. The second, called *leaf-identifying compaction*, allows special types of node-splitting and the last, called *generalized-substitutability-based compaction*, uses a generalized sibling-substitution criterion to achieve further gain. The three algorithms have the same basic top-level pseudocode (Fig. 3) but differ in the implementation of mark-edges and build-result.

A. Basic Compaction

Basic compaction (B-compaction) is designed to conservatively avoid sibling-substitutions that may cause node-splitting. In particular, B-compaction applies sibling-substitution to a node only when an out-going edge e is related to the DC-leaf

[3]General-substitutability is more general than one-sided match proposed by Shiple *et al.* [19] because one-sided match requires $ff_{\mathrm{dc}} \subseteq gg_{\mathrm{dc}}$ to substitute ff by gg even if gg is a cover of ff.

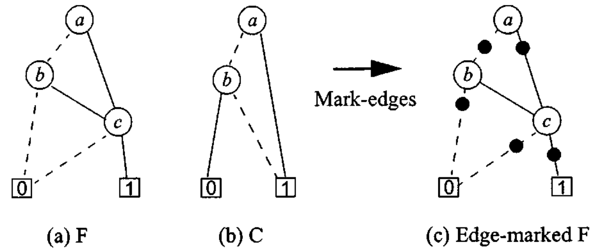


Fig. 4. An example of mark-edges.

```
void mark-edges (bdd f, bdd c) {
    if (c == 0) return;
    if (f == leaf) return;
    x = top variable(f, c);
    if (c_x != 0)
        if (f != f_x) f.then_mark = 1;
        mark-edges(f_x, c_x);
    if (c_x̄ != 0)
        if (f != f_x̄) f.else_mark = 1;
        mark-edges(f_x̄, c_x̄);
}
```

Fig. 5. Mark-edges pseudocode.

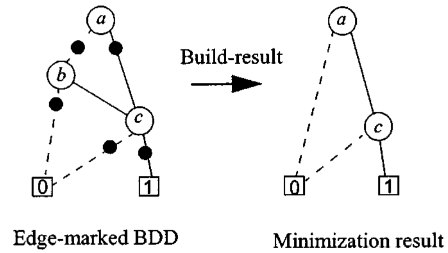


Fig. 6. An example of build-result.

only, i.e., $R(e) = \mathrm{DC}$. To do this, B-compaction *marks* edges that are related to a non-DC node in the mark-edges preprocessing phase and then selectively performs sibling-substitution only on the source nodes of nonmarked edges in the build result phase.

Consider an edge between nodes u and v in F that is related to multiple nodes in C. If any of these nodes is not leaf-0, we can conservatively assume that substituting v with its sibling may cause node-splitting (i.e., node v or a modified version of it is needed). Consequently, mark-edges marks an edge if it is related to anything other than leaf-0. For example, in Fig. 4, the else-edge of c in F is related to both leaf-0 and leaf-1 in C and therefore it is marked. The pseudocode for mark-edges is shown in Fig. 5.

The second build-result phase creates minimized BDD solely based on the markings on edges in F. If an edge from a node v to any of its child nodes u is not marked, then v can be safely replaced by u's sibling via sibling-substitution. Otherwise, v is preserved and its children are recursively rebuilt. Fig. 6 illustrates this *selective* sibling-substitution based rebuilding technique on an edge-marked BDD.

Fig. 7 shows the pseudocode of the build-result routine. For example, if *f.then_mark* is 1 and *f.else_mark* is zero, that means the sub-BDD rooted at f_x must remain (in its original or modified form) and the sub-BDD rooted at $f_{\bar{x}}$ is replaced by its

```
bdd build-result(bdd f) {
    if (f == leaf) return(f);
    x = top variable(f);
    if (f.then_mark == 1 and f.else_mark == 0)
        return (build-result(f_x));
    else if (f.then_mark == 0 and f.else_mark == 1)
        return (build-result(f_x̄));
    else /* if (f.then_mark == 1 and f.else_mark == 1) */
        return (bdd-find(x, build-result(f_x), build-result(f_x̄)));
```

Fig. 7. Build-result pseudocode.

```
bdd B-compaction (bdd f, bdd c) {
    if (c == 0) return (0);
    mark-edges(f, c);
    result = build-result(f);
    clear-edges(f);
    return(result);
}
```

Fig. 8. Basic compaction pseudocode.

sibling in the result. A symmetric rule applies to the reverse case. Note that mark-edges marks at least one of then-edge and else-edge for each node and, thus, does not need to consider the case in which both edges are not marked.

Fig. 8 presents the pseudocode of B-compaction. The time complexity of mark-edges is $O(|F| \cdot |C|)$ because each pair of nodes from F and C is called only once by using an operation cache. Due to the application of a second operation cache, build-result processes each node only once, yielding a time complexity of $O(|F|)$. *Clear-edges* routine clears the edge-marking fields after building the result and has time complexity of $O(|F|)$. Consequently, the overall time complexity of B-compaction is $O(|F| \cdot |C|)$, the same complexity as restrict.

We show that B-compaction is *safe*. Recall that a BDD minimization using DC's is safe if the minimized BDD is guaranteed to be no larger than the original BDD, i.e.,$|F| \geq |F'|$.

Theorem 1: B-compaction is safe.

Proof: The result G of B-compaction on F is produced by build-result. Hence, G results from F by replacing some nodes with one of their descendents.

Intuitively, B-compaction is safe because it ensures that no nodes will be split. This property can be deduced from the structure of build-result. It creates one node for each node it visits (which uniquely depends on the edge-marking) and visits each node at most once (because of the operation cache). Specifically, nodes that are not reachable from the root by a path of marked edges are not visited by build-result and, thus, not included in the minimized BDD.

B. Leaf-Identifying Compaction

This subsection presents an enhanced safe minimization technique in which a special type of node splitting is allowed. Consider the set of sibling-substitutions applied to v to substitute its child u with another child of v. When the results of all the substitutions for u are unique, then the sibling-substitutions can increase the BDD size only by the size of the unique result. Leaf nodes are special in that they are *essential* for all nontrivial BDD's. So, the idea of the new algorithm is to accept the result of sibling-substitution if the result is a unique leaf (i.e., replace the edge from v to u with an edge from v to the leaf). Note that, u may be preserved or replaced in the minimized BDD if it has multiple parents, depending on sibling-substitutions with respect to its other parents.

In effect, more sibling-substitution is allowed in LI-compaction compared to B-compaction. That is, LI-compaction allows sibling-substitution to a node u to replace a child v by its sibling if $R(e) = \{\text{leaf-0}\}$ (like B-compaction does) or to replace v by a leaf if the leaf is a cover of $[v, c]$ for all $c \in R(e)$, where e is the edge between u and v.

This approach will usually lead to better results for two reasons. First, a sub-BDD that is preserved in B-compaction can be replaced by a leaf in LI-compaction. We call this type of gain *Gain 1*. Second, the number of edges marked can be less than in B-compaction because the edge-marking routine needs not recur through edges to be redirected to leaves. This type of gain is called *Gain 2*. Typically, fewer edge-markings leads to smaller BDD's because build-result removes nodes connected by unmarked edges. Note, however, that this approach is not guaranteed to produce better results than B-compaction because the two algorithms can result in different unshared nodes becoming shared unpredictably.

This new approach can be implemented using a two-phase edge-marking routine and a modified build-result. The first phase of edge-marking computes the results of all possible sibling-substitutions from which it identifies the edges that can be redirected to leaves. The second phase is similar to the basic mark-edges routine except that it does not recur through edges that can be redirected to leaves. After the edge-marking, the modified build-result routine redirects all identified edges to their annotated leaf and applies sibling substitution to all remaining unmarked edges.

Fig. 9 shows an example where both gains contribute in minimizing the BDD. First, the then-edge from node a and the then-edge from node b_2 can be redirected to the leaf-0 (Gain 1). Consequently the then-edge of node d is unmarked (Gain 2). The modified build-result routine leads to a minimized BDD with two nodes less than the original BDD. In contrast, B-compaction leads to no minimization because the basic edge-marking routine must mark all edges.

The run-time complexity of this approach is almost twice as much as B-compaction because of the two-phase edge-marking routine since each edge-marking phase has $O(|F| \cdot |C|)$ complexity. If we do not pursue the gain from fewer marked edges (Gain 2), it is possible to merge the two phases of edge marking into one. Our experiments suggest that degradation of quality is negligible. We believe this is because it is unlikely that all nodes on the paths leading to an excessively marked edge can be redirected to a unique leaf (so that no marking is required for the edge). Thus, this compromise represents a good performance/run-time tradeoff.

We refer to this enhanced algorithm with the above compromise as *leaf-identifying compaction (LI-compaction)* and it is given in Fig. 10. Finding and annotating nodes is performed in a preprocessing phase called *LI-mark-edges*. Like restrict, this phase recursively performs sibling-substitution. However,

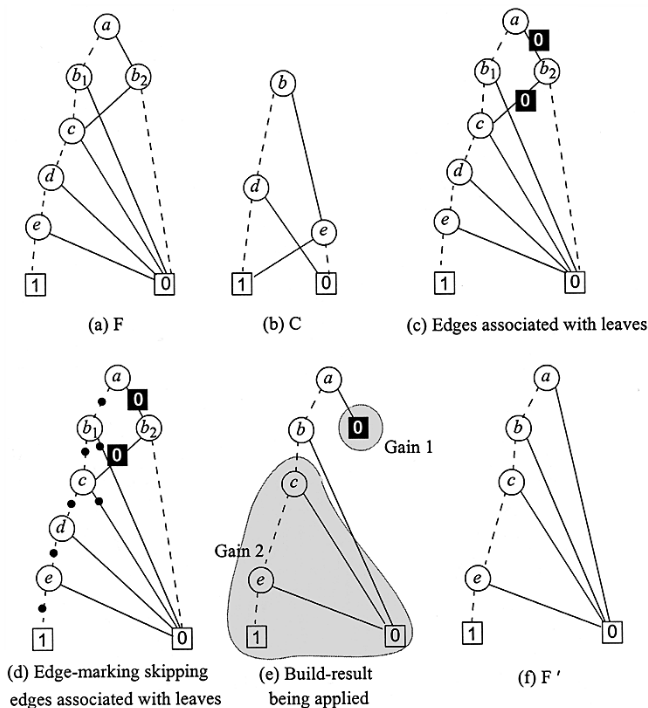


Fig. 9. Improved result by leaf-identification: (a) F, (b) C, (c) edges associated with leaves, (d) edge-marking skipping edges associated with leaves, (d) build-result being applied, and (f) F'.

```
bdd LI-compaction (bdd f, bdd c) {
        if (c == 0) return (0);
        (void) LI-mark-edges (f, c);
        result = LI-build-result (f);
        clear-edges(f);
        return (result);
}
  }
int LI-mark-edges (bdd f, bdd c) {
        if (c == 0) return (00);
        if (f == 1) return (01);
        if (f == 0) return (10);

        x = top variable(f, c);
        temp1 = LI-mark-edges (f_x, c_x);
        temp2 = LI-mark-edges (f_x̄, c_x̄);

        if (f != f_x)
                f.then_mark = f.then_mark | temp1; /* '|' is bitwise-or */
                f.else_mark = f.else_mark | temp2;

        return (temp1 | temp2);
}

bdd LI-build-result(bdd f) {
        if (f == leaf) return (f);
        x = top variable (f);

        if (f.then_mark == 11) f_left = LI-build-result (f_x);
        else if (f.then_mark == 01) f_left = 1;
        else f_left = 0;

        if (f.else_mark == 11) f_right = LI-build-result (f_x̄);
        else if (f.else_mark == 01) f_right = 1;
        else f_right = 0;

        if (f.then_mark == 00 and f.else_mark != 00) return f_right;
        else if (f.then_mark != 00 and f.else_mark == 00) return f_left;
        else return (bdd-find (x, f_left, f_right));
}
```

Fig. 10. LI-compaction pseudocode.

instead of returning the actual BDD result, it returns a classification of the result. This classification identifies whether the edge can be redirected to a 1 (encoded b01), 0 (encoded b10), DC (encoded b00), or nonleaf (encoded b11). The encoding facilitates a bitwise-OR scheme that implements the relative priority of nonleaves over leaves and leaves over DC's. Fig. 11 illustrates an example of leaf-identifying compaction where one edge, the then-edge of d, is additionally marked compared to the example in Fig. 9(d).

The overall time complexity of LI-compaction is the same as the complexity of B-compaction which is $O(|F| \cdot |C|)$.

We would like to note that it is not difficult to identify more *essential* nodes (rather than just leaf nodes) by comparing each F node and its corresponding C nodes. That is, if a node in F corresponds to leaf-1 in C, the node is essential. Consequently, we can extend LI-compaction by allowing sibling-substitutions that returns such essential nodes (instead of just leaves). Our experiments suggest, however, that this extension does not lead to significant improvements over LI-compaction (presumably because the possibility of being able to replace a node with a nonleaf essential node is typically not high). Due to lack of space, we refer the reader to [25] for more details.

C. General-Substitutability Based Compaction

Recall that various criteria can be applied to determine when a node can be replaced by its sibling and we discussed two sibling substitution criteria, DC-substitutability and general substitutability. Fig. 12 illustrates why general substitutability is more powerful than DC-substitutability. Notice that sibling-substitution can be applied to nodes a or b_1 by general-substitutability, however, only b_1 is substitutable by DC-substitutability. The resulting BDD produced by the sibling-substitution applied to a, illustrated in Fig. 12(d), is smaller than obtained by substituting node a, illustrated in Fig. 12(c). The primary reason for this size difference is that node a corresponds to a larger sub-BDD than node b and thereby its sibling substitution typically removes more nodes. Consequently, the application of general substitutability provides more opportunities for sibling substitution than possible using DC-substitutability and, if wisely applied, can lead to smaller BDD's.

We present the pseudocode for the substitutability check based on general-substitutability in Fig. 13. When two nodes can substitute each other, we give priority to substituting the node at the higher level because, as mentioned previously, this typically yields smaller BDD's. Alternatively, we can measure the size of each BDD to find a larger one, but in our experience this does not lead to significant improvements.

To incorporate the generalized criterion into B-compaction, we could simply change the sibling-substitution criteria from DC-substitutability to general substitutability in the edge-marking routine. However, this naive approach fails and the reason is as follows. Consider the F and C BDD's illustrated in Fig. 14(a) and (b), respectively. The else-edge of node b (connected to node c) in F is related to care nodes c_1, c_2. Let's first apply edge-marking using the care node c_1 based on general substitutability. The node c is substitutable

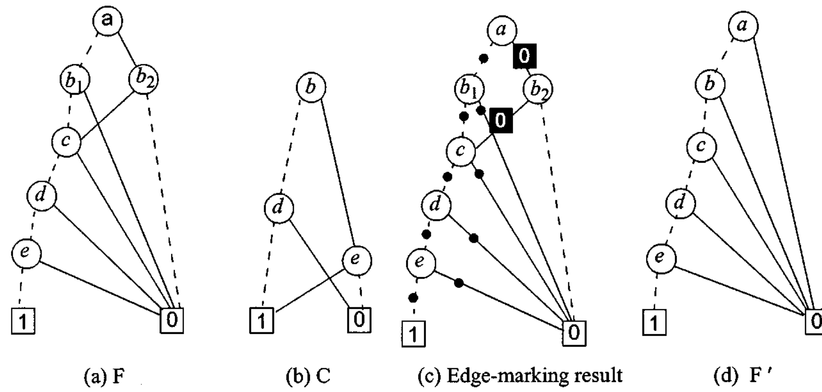


Fig. 11. LI-compaction example: (a) F, (b) C, (c) edge-marking result, and (d) F'.

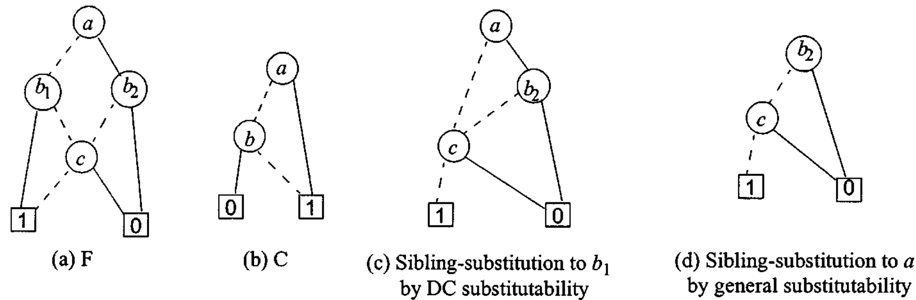


Fig. 12. DC-substitutability versus general-substitutability: (a) F, (b) C, (c) sibling substitution to b_1 by DC substitutability, and (d) sibling substitution to a by general substitutability.

```
int substitutability (bdd f, bdd c) {
    x = top variable(f, c);
    if (c_x == 0) return (f_x to f_x̄);
    if (c_x̄ == 0) return (f_x̄ to f_x);
    fdiff = xor (f_x, f_x̄);

    temp1 = intersect (fdiff, c_x)
    temp2 = intersect (fdiff, c_x̄)

    if (temp1 == ∅ & temp2 == ∅)
        if (root level of f_x > root level of f_x̄) return (f_x to f_x̄);
        else return (f_x̄ to f_x);
    else if (temp1 == ∅) return (f_x to f_x̄);
    else if (temp2 == ∅) return (f_x̄ to f_x);
    else return (NONE);
}
```

Fig. 13. Substitutability check pseudocode (a constant f_x *to* $f_{\bar{x}}$, $f_{\bar{x}}$ *to* f_x, or NONE is returned according to substitutability direction).

by its sibling in this case because they differ only when c is 0 which is in the don't care set specified by c_1. Consequently, the else-edge of b is not marked and using the naive edge-marking procedure we do not recur through b. Thus, at this point all edges below c are not marked [see Fig. 14(c)]. This makes sense because the entire sub-BDD rooted at node c at this point seems unnecessary since we are under the assumption that node c will be removed by a sibling substitution to node b. This assumption, however, breaks down when mark-edges processes the else-edge of node b with its other corresponding care node c_2 which prohibits sibling substitution to node b. The consequence of this assumption breakdown is that mark-edges does not mark some edges that should be marked. In our example, mark edges fails to mark the then-edge of c which should be marked because the sibling-substitution to node b cannot be performed and this edge relates to leaf-1 [see Fig. 14(e)]. This leads to an incorrect cover of F as shown in Fig. 14(f).

This example suggests that we can safely skip recurring mark-edges through an edge *only* when we are guaranteed that the intended sibling-substitution will be performed as assumed. In order to guarantee that the intended sibling-substitution on a node is performed, we must analyze all of its ancestors in conjunction with all of their associated care nodes. This motivates processing the nodes in F in a top-down level-by-level order.

We can process F edges while traversing F from top to bottom level by level. However, this requires cumbersome bookkeeping of all related nodes for each edge. Alternatively, we can accomplish this top-down processing using a two-phase mark-edges routine without increasing run-time complexity. The basic idea is that one phase performs edge-marking assuming that currently unmarked edges will stay unmarked. The other phase checks if the assumption associated with each sibling-substitution (that a particular edge remains unmarked) holds and reinvokes the first edge-marking phase as required if the assumption is invalidated.

The pseudocode for this new compaction algorithm called *generalized substitutability based compaction (GS-compaction)* is presented in Fig. 15. The first edge-marking phase, called *mark-essential-edges*, tests the sibling-substitution condition only if the edge connected to the node considered has not been marked thus far. If the substitutability holds, it does not recur through the node to be substituted, marks the edge connected to

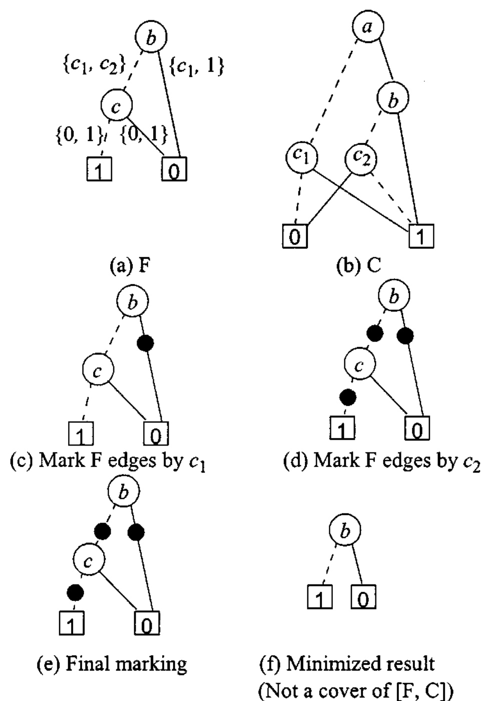


Fig. 14. Failure of using general-substitutability in B-compaction: (a) F, (b) C, (c) mark F edges by c_1, (d) mark F edges by c_2, (e) final marking, and (f) minimized result (not a cover of $[F, C]$).

its sibling, and continues this process to the node connected by the marked edge. The sibling-substitution is recorded in a list whose elements are triples of the node in F, the node in C, and the sibling-substitution direction. For the example presented in Fig. 14 the marking-result shown in Fig. 14(e) is the result of *mark-essential-edges*. The second edge-marking phase, called *mark-supplemental-edges*, sorts the list by the level of F nodes in order to process the edges connected from higher F nodes first. Then, it removes the first sibling-substitution, e.g., substituting v in F with its sibling w, from the list and check if it is valid. If the sibling-substitution is still valid, the care-set of v needs to be added to the care-set of w. This care-set update is implicitly done by invoking mark-essential-edges with w and the node representing the care-set of v as parameters. If the sibling-substitution of v is found invalid, mark-essential-edges is invoked with v and the node representing the care-set of v as parameters. When *mark-essential-edges* finishes, *mark-supplemental-edges* resumes the validation process until the list is empty. Note that new sibling-substitutions might be recorded by *mark-essential-edges* which is invoked by *mark-supplemental-edges*. Build-result routine is the same as in B-compaction. In our example, the sibling-substitution to b in F is checked first. As both outgoing edges are found marked, the sibling-substitution is invalidated and then-edge of c in F becomes marked eventually.

The time complexity of mark-essential-edges is $O((|F| \cdot |C|)^2)$ because the substitutability routine requires $O(|F| \cdot |C|)$ and that routine is called at most $|F| \cdot |C|$ times by mark-essential-edges. The time complexity of mark-essential-edges is $O((|F| \cdot |C|)^2 + |F| \cdot |C| \cdot \log(|F| \cdot |C|))$, where the first term describes the time complexity of mark-essential-edges and the second term describes the time complexity

```
bdd GS-compaction (bdd f, bdd c) {
        if (c ==0) return (0);
        fcs_list = create-list();
        mark-essential_edges(f, c, fcs_list);
        mark-supplemental-edges(fcs_list);
        clear-edges(f);
        return (build-result(f));
}

void mark-essential-edges (bdd f, bdd c, list fcs_list) {
        if (c == 0  || f == leaf) return;
        x = top variable(f, c);
        if (f != fx)
                s = substitutability (f, c);
                if (f.then_mark== 0)
                        if (s ==fxtofx) insert(fcs_list, {f, c, s});
                        else f.then-edge = 1;
                if (f.else_mark== 0)
                        if (s = fxtofx) insert(fcs_list, {f, c, s});
                        else f.else-edge = 1;
        if (f.then_mark == 1 || f==fx) mark-essential-edges(fx, cx, fcs_list);
        if (f.else_mark  == 1 || f==fx) mark-essential-edges(fx, cx, fcs_list);
}

void mark-supplemental-edges (list fcs_list) {
        while (fcs_list not empty) {
                sort fcs_list by the level of f in each triple;
                fcs = first(fcs_list);
                remove (fcs, fcs_list);
                f = fcs.f; c = fcs.c; s = fcs.s;
                x = top variable(f, c);
                if (s == fxtofx)
                        if (f.then_mark == 0) mark-essential-edges(fx, cx, fcs_list);
                        /* fx is guaranteed to substitute fx. Include cx to the care sets of fx */
                        else mark-essential-edges(fx, cx, fcs_list);
                        /* fx cannot substitute fx. Do edge-marking that was skipped. */

                if (s == fxtofx)
                        if (f.else_mark == 0) mark-essential-edges(fx, cx, fcs_list);
                        /* fx is guaranteed to substitute fx. Include cx to the care sets of fx */
                        else mark-essential-edges(fx, cx, fcs_list);
                        /* fx cannot substitute fx. Do edge-marking that was skipped. */
        }
}
```

Fig. 15. GS-compaction pseudocode.

for quick sort. Note that mark-essential-edges does not repeat the same computation because of the operation cache. Build-result and clear-edges both have $O(|F| \cdot |C|)$ time complexity. Therefore, the overall time complexity of GS-compaction is $O((|F| \cdot |C|)^2)$.

D. Multiple BDD's Minimization

It is important to note that safe BDD minimization does not itself guarantee overall reduction in the size of multioutput circuits. This is because each output is represented by one BDD and existing safe BDD minimization does not consider the sharing among BDD's. Consequently, minimizing one BDD may reduce the sharing among BDD's, potentially leading to an overall increase in BDD nodes. This suggests that the synthesized circuit might be larger after BDD minimization using existing techniques.

Fortunately, we can extend the concept of safety to handle multiple BDD's. The basic idea is to first complete all edge-markings for each output function f and corresponding care set c, and only then apply build-result for each f. It is easy to show that this simple modification ensures that minimization of multiple BDD's will be safe. In fact, for circuits with lots of sharing

TABLE I
MINIMIZATION RESULTS ON BDD's FOR SEQUENTIAL CIRCUITS USING UNREACHABLE STATES AS DC's

circuit	BDD_f	BDD_c		reduced BDD_f size (reduction runtime)					improv.
	size	size	on-set (%)	restrict	osm_bt	B	LI	GS	(%)
s298	183	57	1.3	137 (0.01)	129 (0.06)	149 (0.38)	140 (0.37)	118 (1.08)	9.3
s344	260	610	8.0	253 (0.15)	248 (1.00)	253 (0.62)	253 (0.61)	235 (1.62)	5.5
s382	262	304	0.4	245 (0.04)	231 (0.35)	265 (0.54)	245 (0.53)	200 (1.54)	15.5
s444	300	173	0.4	218 (0.12)	205 (0.32)	261 (0.54)	224 (0.53)	193 (1.56)	6.2
s499	1073	72	5.2e-5	340 (0.05)	340 (0.11)	340 (0.85)	340 (0.54)	272 (2.42)	25
s510	281	8	73.4	265 (0.01)	262 (0.04)	269 (0.26)	265 (0.27)	264 (0.72)	-1.0
s526	301	125	0.4	251 (0.03)	220 (0.13)	254 (0.53)	251 (0.52)	204 (1.51)	7.8
s641	884	83	0.3	607 (0.06)	577 (0.86)	599 (0.85)	599 (0;85)	548 (2.48)	5.3
s820	499	8	78.1	472 (0.02)	471 (0.07)	469 (0.47)	469 (0.45)	468 (1.39)	0.6
s953	867	562	1.6	739 (0.04)	739 (0.39)	751 (0.97)	739 (0.99)	712 (2.78)	3.8
s1196	3974	919	1.4	3974 (0.49)	3971 (10.5)	3974 (0.93)	3974 (0.95)	3966 (3.05)	0.1
s1269	40487	949	1.1	29699 (2.71)	29724 (222)	28000 (6.63)	27995 (6.55)	27968 (25.5)	6.2
s1423	37269	66	1.3	30610 (1.50)	30974 (1084)	30611 (4.80)	30610 (4.50)	30580 (19.9)	0.1
s3271	2917	417	0.8	2787 (0.74)	2733 (29.5)	2793 (10.6)	2781 (11.0)	2629 (21.0)	4.0
total				70597 (5.97)	70824 (1349)	68988 (29.0)	68885 (28.7)	68357 (240)	6.3

among cones of logic this feature can have a significant impact in overall BDD size.

We can apply this separated edge-marking and build-result concept to all three proposed compaction heuristics. In our experiments, however, we only applied this method to GS-compaction. We call this variant of GS-compaction GSM-compaction.

IV. EXPERIMENTAL RESULTS

We conducted experiments in SIS-1.2 [26] to compare our heuristics to existing sibling-substitution based heuristics, specifically, restrict and one of its variants *osm-bt* [19]. Osm-bt was chosen among a variety of heuristics developed by Shiple *et al.* [19] because it showed the best overall results in the examples they tested. Because the two existing heuristics can produce larger BDD's than the original BDD's, we applied thresholding to them which means that we return the original BDD's if the heuristics produce larger BDD's. All the experiments were conducted on a SUN SPARC 20/128 MB.

In our first experiment, we minimized the BDD's representing the combinational logics of sequential circuits from ISCAS-89 and ISCAS-Addendum-93 benchmark circuits using their unreachable states as DC's. The BDD variables were ordered using the static variable ordering heuristic proposed in [22].[4] For some of the largest circuits, exact reachable states could not be computed because the memory requirements were too high. For three of these circuits, however, we were able to compute a superset of reachable states using *machine-by-machine* (MBM) traversal, an approximate finite state machine (FSM) traversal technique proposed by Cho *et al.* [27], and used the complement of the result as DC's. The result of MBM traversal takes the form of implicit conjunction, i.e., a set of BDD's representing reachable states on partitioned state spaces, and we constructed a single care BDD by conjuncting all of them. We minimized each BDD representing a combinational logic cone using the single care BDD, individually.

The results are given in Table I. The nodes shared by multiple BDD's are counted once for each BDD to better illustrate the minimization quality on individual BDD's. GS-compaction demonstrates the best performance except for one example s510 in which the difference is only 2 nodes. This suggests that the safety feature consistently improves minimization quality when combined with general substitutability. We show the ratio of the smallest BDD sizes obtained using new heuristics and existing heuristics in the column denoted *improv*. New heuristics produce up to 15.5% (6.3% on average) smaller BDD's than the BDD's produced by existing heuristics. We observed that there were many BDD's that no heuristics can reduce their sizes at all (possibly because those BDD's were already minimized). When we exclude such BDD's from the result, the improvement factor becomes almost two times larger.

The numbers in parentheses in the Table I indicate the run-time of each minimization in units of CPU seconds. Because compaction heuristics require more phases than restrict, they are slower than restrict. In particular, GS-compaction requires a time-consuming substitutability check which requires significant run-time overhead. However, GS-compaction is much faster than its counterpart osm-bt for large BDD's even though GS-compaction uses a more general substitution criterion than the one used by osm-bt. This is because GS-compaction skips many substitutability checks, i.e., GS-compaction does not check substitutability for the nodes connected by marked edges.

In our second experiment, we minimized the BDD's representing the combinational circuits from ISCAS-89 and MCNC-91 benchmark. The BDD variables were statically ordered using the heuristic proposed in [2]. We used randomly created DC's with 95% and 5% DC fractions for each BDD representing a combinational logic cone to demonstrate the

[4]Performing dynamic variable ordering (DVO) on the original and/or minimized BDD's can sometimes achieve further reductions in BDD size (at the expense of significant additional run-time) but was not considered here to keep the experiment and the analysis of the results simple and focused.

TABLE II
MINIMIZATION RESULTS ON BBD'S FOR COMBINATIONAL CIRCUITS WITH RANDOMLY GENERATED 95% DC'S

circuit	BDD_f size	BDD_c size	reduced BDD_f size (reduction time)						improv. (%)
			restrict	osm_bt	B	LI	GS	GSM	
bw	112	40	48.0 (0.01)	48.0 (0.02)	48.0 (0.50)	48.0 (0.46)	48.0 (1.28)	50.3 (0.05)	0.00
misex3c	797	359	251.2 (0.06)	221.6 (0.48)	306.0 (0.32)	280.0 (0.32)	219.2 (0.91)	229.2 (0.27)	1.09
duke2	973	1559	474.0 (0.07)	399.0 (0.72)	561.3 (0.59)	409.7 (0.61)	295.0 (1.66)	290.0 (0.34)	37.5
vg2	1044	5028	442.0 (0.09)	419.6 (0.21)	419.5 (0.20)	381.5 (0.21)	283.0 (0.70)	283.9 (0.34)	48.3
misex3	1301	359	471.0 (0.04)	436.3 (0.48)	581.1 (0.34)	505.1 (0.32)	414.0 (0.92)	412.8 (0.22)	5.70
C432	1733	202	1673.2 (0.01)	1673.2 (0.01)	1640.0 (0.24)	1634.8.9 (0.10)	1612.8 (0.22)	1581.5 (0.06)	3.70
alupla	17266	21699	17482.0 (3.7)	15190.7 (78.9)	15609.0 (2.87)	15188.0 (2.85)	6667.5 (15.1)	6562.9 (15.0)	131.5
C1908	36007	4490	31640.1 (21.0)	timeout	31264.8 (16.3)	32546.8 (16.5)	31822.8 (74.1)	29494.2 (83.8)	7.28
C499	45922	726	7566.0 (44.0)	timeout	7880.9 (36.7)	9663.0 (39.0)	9031.4 (234)	4354.4 (380)	73.8
seq	142252	7763	89880.0 (8.28)	82516.5 (302)	79330.0 (14.0)	77637.0 (14.3)	67157.5 (62.1)	67949.0 (78.2)	22.9
average									33.2

TABLE III
MINIMIZATION RESULTS ON BBD'S FOR COMBINATIONAL CIRCUITS WITH RANDOMLY GENERATED 5% DC'S

circuit	BDD_f size	BDD_c size	reduced BDD_f size (reduction time)						improv. (%)
			restrict	osm_bt	B	LI	GS	GSM	
bw	112	40	92.0 (0.01)	87.0 (0.01)	92.2 (0401)	92.2 (0.45)	87.0 (1.30)	91.0 (0.05)	0.0
misex3c	797	359	759.2 (0.01)	757.4 (0.39)	785.1 (0.32)	755.1 (0.31)	739.9 (0.92)	751.3 (0.21)	2.37
duke2	973	1559	956.5 (0.04)	951.5 (1.01)	966.7 (0.62)	953.7 (0.62)	932.0 (1.78)	932.0 (0.28)	2.09
vg2	1044	5028	1050.0 (0.07)	1050.0 (3.12)	1048.3 (0.38)	1048.3 (0.35)	1048.3 (0.65)	1030.2 (0.34)	1.92
misex3	1301	359	1301.0 (0.05)	1301.0 (0.77)	1291.2 (0.35)	1288.9 (0.39)	1300.7 (1.05)	1301.8 (0.78)	0.94
C432	1733	202	1734.0 (0.16)	1734.0 (44.0)	1734.0 (0.39)	1734.0 (0.32)	1732.0 (1.56)	1731.5 (1.3)	0.14
alupla	17266	21699	17268.1 (2.30)	17268.1 (117)	17266.3 (3.12)	17266.2 (3.10)	17257.2 (12.9)	17258.4 (12.7)	0.06
C1908	36007	4490	36612.5 (22.3)	timeout	36005.1 (17.4)	36053.2 (18.7)	36031.2 (67.5)	35801.4 (69.2)	2.27
C499	45922	726	45922.0 (52.8)	timeout	45922.0 (48.1)	45922.0 (49.0)	45922.0 (439)	45922.0 (431)	0.0
seq	142252	7763	142150.2 (5.40)	141590.0 (956)	138566.2 (16.9)	138384.0 (18.0)	137991.0 (65.1)	137962.3 (64.4)	2.63
average									1.24

impact of the DC fraction on the improvement ratio. As suggested in [23], [28], and [29], to increase the statistical significance of our results, for each DC fraction, we used five different seeds to generate 5 different DC sets and report the average minimization ratios. In particular, for each DC set, we minimized each BDD representing a combinational logic cone using its respective care BDD with all the minimization heuristics. Recall that GSM-compaction, the modified version of GS-compaction which safely minimizes multioutput BDD's, however, performs edge-markings on all BDD's before building the result. The results from the modified version of other compaction heuristics are not shown because they show similar improvements compared to their original versions.

Tables II and III show the results with 95% DC's and 5% DC's, respectively. We count the nodes shared by multiple BDD's only once for these results. The results show that our compaction algorithms consistently outperform the competitive algorithms in all cases. The improvement factors for our compaction algorithms, however, are much higher in the case of 95% DC fractions. This is because there is more flexibility of minimization with more DC's. Second, the results show that the additional safety feature of GSM-compaction for multi-output BDD's can lead to dramatic reductions in BDD size. For example, the BDD's obtained using GSM-compaction for C499 are more than two times smaller than the BDD's obtained using GS-compaction. However, it is also interesting to note that GSM-compaction does not always produce better results than GS-compaction because sometimes node splitting leads to smaller BDD's whose total size is smaller than its originally shared BDD. (Recall that this is the same reason why safe BDD minimization might not be better than nonsafe heuristics.)

The advantage of GSM-compaction over osm-bt in terms of run-time is clearer in this experiment because we observe two examples that osm-bt cannot complete the minimization within 10 h.

V. CONCLUSION

We presented new efficient heuristics to minimize the size of BDD's using DC's. The key idea of new heuristics is to selectively minimize sub-BDD's while traditional heuristics blindly minimize all sub-BDD's. By removing the source of size growth, we were able to achieve better overall minimization quality. We demonstrate that the new heuristics significantly outperform traditional heuristics on most examples from benchmark circuits. We also generalize our mechanism to prevent the size growth to handle multiple BDD's with sharing and present experimental results that demonstrate its effectiveness.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of this work. They would also like to thank Dr. R. Drechsler of the Albert–Ludwigs-University Freiburg for valuable discussions and feedback on an earlier version of this work.

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, 1986.

[2] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. Int. Conf. Computer-Aided Design*, 1988, pp. 6–9.

[3] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi, "Variable ordering for FSM traversal," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 476–479.

[4] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 472–475.

[5] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 42–47.

[6] B. Lin and S. Devadas, "Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams," in *Proc. Int. Conf. Computer-Aided Design*, 1994, pp. 542–549.

[7] K. Y. Yun, B. Lin, D. Dill, and S. Devadas, "Performance-driven synthesis of asynchronous controllers," in *Proc. Int. Conf. Computer-Aided Design*, 1994, pp. 550–557.

[8] T. Karoubalis, G. P. Alexiou, and N. Kanopoulos, "Optimal synthesis of differential cascode switch logic circuits using ordered binary decision diagrams," in *Proc. European Design Automation Conf.*, 1995, pp. 282–287.

[9] L. Lavagno, P. McGeer, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "Timed Shannon circuits: A powerful design style and synthesis tool," in *Proc. Design Automation Conf.*, 1995, pp. 254–260.

[10] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *Proc. Design Automation Conf.*, 1990, pp. 620–625.

[11] S. Chang, M. Marek-Sadowska, and T. Hwang, "Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1226–1236, Oct. 1997.

[12] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Proc. Design Automation Conf.*, 1995, pp. 587–592.

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

[14] Y. Hong, P. A. Beerel, L. Lavagno, and E. M. Sentovich, "Don't care-based BDD minimization for embedded software," in *Proc. Design Automation Conf.*, 1998, pp. 506–509.

[15] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size for incompletely specified functions," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1435–1437, Nov. 1996.

[16] A. L. Oliveira, L. Carloni, T. Villa, and A. Sangiovanni-Vincentelli, "Exact minimization of Boolean decision diagrams using implicit techniques," Univ. California, Tech. Rep. UCB ERL M96/16, 1996.

[17] R. Drechsler, N. Drechsler, and W. Gunther, "Fast exact minimization of BDD's," in *Proc. Design Automation Conf.*, 1998, pp. 200–205.

[18] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Automatic Verification Methods for Finite State Systems*. Berlin, Germany: Springer-Verlag, 1989, pp. 365–373.

[19] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton, "Heuristic minimization of BDD's using don't cares," in *Proc. Design Automation Conf.*, 1994, pp. 225–231.

[20] R. Drechsler and N. Göckel, "Minimization of BDD's by evolutionary algorithms," in *Proc. Int. Workshop Logic Synthesis*, 1997.

[21] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *Proc. Int. Conf. Computer-Aided Design*, 1990, pp. 126–129.

[22] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *Proc. Int. Conf. Computer-Aided Design*, 1990, pp. 130–133.

[23] S. Chang, D. I. Cheng, and M. Marek-Sadowska, "Minimizing ROBDD size of incompletely specified multiple output functions," in *Proc. European Design and Test Conf.*, 1994, pp. 620–624.

[24] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[25] Y. Hong and P. A. Beerel, "Improving the quality of safe BDD minimization using don't cares," Univ. Southern California, Tech. Rep. CENG 97-14, 1997.

[26] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, 1992, pp. 328–333.

[27] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for approximate FSM traversal," in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 25–30.

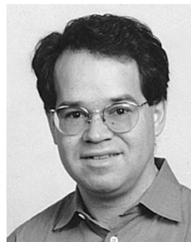
[28] J. E. Harlow III and F. Brglez, "Design of experiments in BDD variable ordering: Lessons learned," in *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 646–652.

[29] K. Wang and T. Hwang, "Boolean matching for incompletely specified functions," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 160–168, Feb. 1997.

Youpyo Hong (S'92–M'98) received the B.S. degree in electrical engineering from Yonsei University, Seoul, Korea, in 1991, and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California, Los Angeles, in 1993 and 1998, respectively.

He was with Synopsys, Beaverton, OR, from 1998 to 1999, where he worked on formal verification. Since 1999, he has been with the Electronical Engineering Department of Dongguk University, Seoul, Korea, as a faculty member. His current work focuses on logic synthesis and formal verification.

Dr. Hong is a member of Phi Kappa Phi.



Peter A. Beerel (S'88–M'95) received the B.S.E. degree in electrical engineering from Princeton University, Princeton, NJ, in 1989 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1991 and 1994, respectively.

He is an Assistant Professor of the Electrical Engineering-Systems Department at the University of Southern California (USC), Los Angeles.He was a primary consultant for Intel Corporation on their Asynchronous Instruction Length Decoder Project.He has been a member of the Technical Program Committee of the International Symposia on Advanced Research in Asynchronous Circuits and Systems (ASYNC) since 1997, and was Program Cochair for ASYNC'98. His research interests include CAD, formal verification, and mixed asynchronous/synchronous VLSI systems.

Dr. Beerel won the Junior Research Award of the USC School of Engineering in 1998 and was a recipient of the USC School of Engineering's Outstanding Teaching Award in 1997. He was also cowinner of the Charles E. Molnar award for two papers published in ASYNC'97 that best bridged theory and practice of asynchronous system design and was cowinner of the Best Paper Award in ASYNC'99. He received a National Science Foundation (NSF) Career Award and a 1995 Zumberge Fellowship.

Jerry R. Burch received the B.S. and M.S. degrees in computer science from the California Institute of Technology in 1984 and 1985, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University , Pittsburgh, PA, in 1992.

He is currently a Research Scientist at Cadence Berkeley Laboratories, Berkeley, CA. His research interests include formal verification, binary decision diagrams, technology mapping, and models of concurrent systems. He has published more than 20 technical papers and has developed prototype tools for automatically verifying several different classes of circuits, including pipelined microprocessors.

Kenneth L. McMillan received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1992.

Currently a Research Scientist at Cadence Berkeley Laboratories, Berkeley, CA, his research interests are in various aspects of formal verification, especially for hardware design. His current work centers on compositional methods for hardware design. He is the author of *Symbolic Model Checking* (Norwell, MA: Kluwer Academic, 1993).

Dr. McMillan received the ACM Doctoral Dissertation Award for his Ph.D. dissertation on symbolic model checking, and the 1998 ACM Paris Kannelakis Award.