A Simple BDD Package without Variable Reordering and Its Application to Logic Optimization with Permissible Functions

Yukio Miyasaka[1], Alan Mishchenko[2], Masahiro Fujita[1]
[1]University of Tokyo
[2] University of California, Berkeley
E-mail: miyasaka@cad.t.u-tokyo.ac.jp

Abstract

Binary Decision Diagrams (BDDs) are a canonical representation of logic functions, which allows for efficient computation in logic verification and optimization. Although the BDD size is sensitive to the variable ordering, reasonably good orderings can be found for many functions in hardware design. For other functions, such as multipliers, the BDD size is exponentially large for any ordering. On the other hand, dynamic reordering can significantly reduce the BDD size, but it is often time- and memory-consuming.

This paper proposes a new simple BDD package without dynamic variable reordering, which is much faster than the conventional BDD package with reordering. By avoiding the reordering, a substantial amount of memory (40%) can be saved. The proposed BDD package is used in logic optimization with permissible functions and is applied to benchmarks synthesized from PLA. The results show that the AIG node count for some benchmarks is reduced by 40% at most and 10% on average while conventional logic synthesis tools, such as ABC, largely miss the optimization opportunity.

Keywords

BDDs, CSPF, Formal Verification, Combinational Circuit, Circuit Minimization

1. Introduction

BDDs are a data structure, which represents a logic function canonically if the order of the variables is given [1]. Because of the canonicity, we can prove the equivalence of two circuits by building BDDs for their outputs. The circuits are equivalent if and only if the BDDs for their outputs are the same.

The disadvantage of BDDs is that if the appropriate order of variables is not given, the number of nodes increases exponentially. We can often obtain a good variable ordering, but there are cases when the number of nodes is always exponential, for example, in the case of multipliers [2].

Dynamic variable reordering has been proposed to reduce the number of nodes. It dynamically changes the order of variables while building a BDD. To handle computations of the reordering, it requires much longer time and extra memory to store pointers among the nodes.

In this paper, we implemented a simple BDD package without dynamic reordering to save memory. Because the bottleneck of building BDDs is memory, the reduction in memory usage leads to faster computation.

We also implemented the transduction method [3] based on permissible functions for logic optimization using the simple BDD package. The permissible function represents don't-cares of a node in the network. By managing permissible functions with BDDs, the memory requirements are reduced, and the optimization is performed faster. The transduction method removes unnecessary gates and wires while transforming the circuit using the permissible functions.

In the experiment, we compared the performance of CUDD [4] and the proposed simple BDD package by building BDDs for integer multipliers and some benchmarks. Next, we compared the performance of circuit minimization using ABC [5] and using the proposed BDD-based transduction method.

The paper is organized as follows. Section 2 introduces the new simple BDD package. Section 3 reviews the transduction method implemented using the package. Section 4 describes the experimental setup. Section 5 shows the experimental results. Section 6 concludes the paper.

2. A simple BDD package

2.1. Basic components

It was proved [2] that the number of nodes in the BDD representing an integer multiplier grows exponentially regardless of a variable order. The dynamic reordering is not needed for building BDDs of integer multipliers.

We implemented a simple BDD package without any dynamic reordering. Because pointers connecting nodes during dynamic reordering can be saved, the new BDD package has less memory usage per node. The memory usage per node is shown in Table 1. The specified number of nodes are allocated before building BDDs in this package. Let *nAlloc* be that number.

The variables of BDDs are the inputs of the circuits. They are denoted as "PI-variables" in this section.

A BDD node is identified by a 31-bit BDD-variable and has a PI-variable stored as an 8-bit char, literals of two children (THEN and ELSE) stored as two 32-bit ints, and a multi-purpose mark stored as a 8-bit char. Note the maximum possible number of nodes is 2^{31}.

A literal is a 32-bit int, which is a BDD-variable with a complemented attribute in the LSB. A literal with its LSB 1 represents the complement of the node. The node with BDD-variable 0 is constant-0. The node of the first input of the circuit has BDD-variable 1, its THEN is literal 1, and its ELSE is literal 0. A literal for ELSE must not be complemented to avoid duplication of literals. A node *a* is equivalent to the complement of the node *b* whose PI-variable, THEN, and ELSE are the same PI-variable, the complement of the same THEN, and the complement of the same ELSE as the node *a* respectively.

Table 1: Memory footprint of a node in the simple BDD package

Purpose	Type	Size(Byte)
PI-Variable	char	1
Children	2 × int	8
Unique Table	2 × int	8
Computed Table	3 × int	12
Mark	char	1
Total		30

In order to guarantee the uniqueness of the node, we use a hash table with linked lists. Each linked list stores BDD-variables of the nodes found in the same bin. Before a new node is created, a hash value is calculated using its PI-variable and children. The linked list for the hash value is searched to check whether there is a node whose PI-variable and children are the same as the new node. If such a node is found, that node is used as the new node without actually creating the new node. Otherwise, a new node with a new BDD-variable is created and the new BDD-variable is inserted at the head of the linked list. In our implementation, the size of hash table to store the heads of linked lists is set equal to *nAlloc*. The size of the table to store the next elements in linked lists is also set equal to *nAlloc*. In total, the unique table requires two 32-bit ints for each BDD node.

The new package has a computed table to speed up the BDD construction. Let *x*, *y*, *z*, *p*, and *q* be literals. After building *z* as AND of *x* and *y* ($x < y$), a hash value is calculated using *x* and *y*. Three literals *x*, *y*, and *z* are stored at the address of the hash value in the computed table. Before building a BDD for AND of *p* and *q* ($p < q$), the computed table is checked at the address of hash value calculated from *p* and *q*. If there is an entry whose first and second values are equal to *p* and *q* respectively, its third value is used as the result of AND of *p* and *q*. We set the size of computed table equal to *nAlloc*, while each entry requires three 32-bit ints.

2.2. Building BDDs and garbage collection

For equivalence checking, we build a BDD for each output of the circuit. Because the outputs would have some overlap among their fanin cones, BDDs for all outputs are built at the same time. A BDD is built for each gate in a topological order from inputs to outputs of the circuit. Note a BDD for a gate means a BDD representing a logic function realized at the output of the gate.

The BDD for a gate is no longer necessary after BDDs for all gates in fanout of the gate are built, unless its fanout includes the output of the circuit. It is called a dead BDD. A node in a dead BDD can be removed if the node is not used in any live (non-dead) BDD.

The new package uses the mark/sweep garbage collection to remove the unnecessary nodes. When a BDD for a gate is built, the root node of the BDD is marked. That root node is unmarked after BDDs for all gates in fanout of the gate are built, if the output of the circuit is not in its fanout. Thus, all root nodes of live BDDs are always marked. When sweeping, all descendants of marked nodes are kept, the other nodes are removed.

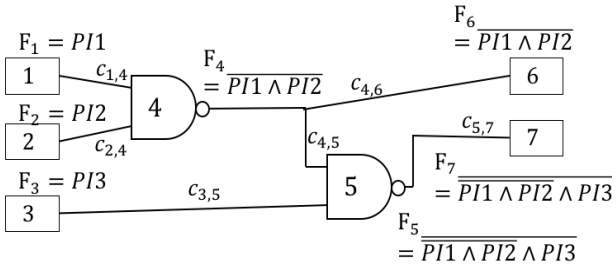

Figure 1: Logic functions in a circuit.

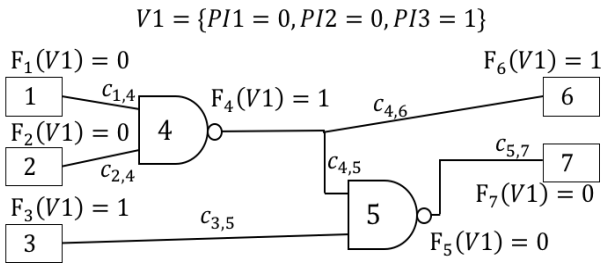

Figure 2: The values of logic functions in a circuit for a PI pattern *V1*.

The sweeping is called when the number of nodes reaches *nAlloc*. The links in the linked lists of unique table are reconnected to skip the removed nodes. The computed table is initialized because updating all entries containing the removed nodes is time consuming. The PI-variable of the removed node is set to -1, and the minimum BDD-variable of removed nodes is memorized while sweeping. The BDD-variable of a new node is set to the minimum BDD-variable of removed nodes after the sweeping. The next minimum BDD-variable of removed nodes can be found by searching the node with PI-variable -1.

If building a BDD for a gate runs out of nodes even after the sweeping, the reallocation is performed with *nAlloc* doubled (*nAlloc*:=2 × *nAlloc*). If *nAlloc* becomes larger than 2^{31}, the BDD construction is terminated. The unique table must be rehashed after the reallocation. The new hash value for a node is whether the same or increased by half of *nAlloc* as the previous one because the hash value is calculated in modulo *nAlloc*. Each linked list is walked and the nodes with the hash value increased are moved to the new linked list, while the other nodes remain in the same linked list.

3. The transduction method using CSPF with BDD

We implemented a circuit minimization method called the transduction method using the simple BDD package. In order to make it easy to apply the transduction method, it is assumed that all gates in the circuit are NAND gates.

Here we explain the terminology in this section. A primary input/output (PI/PO) is an input/output of the circuit. Let F_i be a logic function realized at the output of a gate *i* or at a PI/PO *i* as shown in Figure 1. The inputs of a function are PIs. A PI pattern is a pattern of PIs' values. For a function X, X(*V*) is a value of X for a PI pattern *V*. For example, the value of $F_i(V1)$ for a PI pattern *V1*={*PI1=0, PI2=0, PI3=1*} is shown in Figure 2. $c_{i,j}$ is a connection from a gate or PI *i* to a gate or PO *j*. FI_i is a set of all gates and PIs in fanin of gate *i*

and FO_i is a set of all gates and POs in fanout of gate *i*. In other words, FI_i is a complete set of gate or PI *j* where $c_{j,i}$ exists and FO_i is a complete set of gate or PO *j* where $c_{i,j}$ exists. FIC_i/FOC_i is a set of all gates in fanin/fanout cone of gate *i*.

3.1. The permissible function

The permissible function is a function which outputs 0, 1, or * (don't-care). A permissible function is calculated for each gate and each connection in the circuit. Let G_i be a permissible function at the output of a gate *i* and $G_{i,j}$ be a permissible function at a connection $c_{i,j}$. Both G_i and $G_{i,j}$ are initially set equal to F_i for every gate *i* and every connection $c_{i,j}$. When we find a PI pattern *V* where the output of gate *i* or $c_{i,j}$ is don't-care, $G_i(V)$ or $G_{i,j}(V)$ is made to be *. Note $G_i(V)/G_{i,j}(V)$ is a value of $G_i/G_{i,j}$ for a PI pattern *V*.

In the transduction method, the circuit is transformed in the way that F_i is included in G_i for any gate *i* even after the transformation. Here "G_i includes F_i" means that $F_i(V)$ is equal to $G_i(V)$ for any PI pattern *V* where $G_i(V)$ is not *.

There have been two sets of permissible functions proposed [3]. One is the maximum set of permissible functions (MSPF), the other is the compatible set of permissible functions (CSPF).

MSPF represents a complete set of don't cares. It is calculated the following way. Note a logic operation between functions is performed as a logic operation for every PI pattern. For example, "a function Z is AND of functions X and Y" means Z(*V*) is AND of X(*V*) and Y(*V*) for every PI pattern *V*.

1. For each gate *i* in a topological order from POs to PIs, do 2 and 3.
2. Calculate AND' (Table 2) of $G_{i,j}$ for every gate or PO *j* in FO_i, and the result is assigned to G_i.
3. For each gate or PI *j0* in FI_i, do 4 and 5.
4. Calculate AND of F_{j1} for every gate or PI *j1* in FI_i except *j0=j1*. Let X be the result.
5. Calculate @ (Table 3) of G_i and X, and the result is assigned to $G_{j0,i}$.

2 follows that the output of a gate is don't-care if all output connections of the gate are don't-care. 4 and 5 follow that an input connection of a gate is don't-care if the output of the gate is don't-care or at least one of the other input connections is 0. Otherwise, the input connection must be the complement of the output of the gate. Note a gate is a NAND gate and the value of G_i cannot be 0 if there is an input connection whose value is 0 (X=0).

The defect of MSPF is that we must recalculate the permissible functions after some transformations of the circuit. Consider a gate *i* which has 2 input connections: $c_{j0,i}$ and $c_{j1,i}$. For a PI pattern *V* where both $F_{j0}(V)$ and $F_{j1}(V)$ are 0, both $G_{j0,i}(V)$ and $G_{j1,i}(V)$ are made to be * in MSPF. If $F_{j0}(V)$ changes from 0 to 1 in a transformation of the circuit, we must change $G_{j1}(V)$ from * to 0 and recalculate permissible functions according to that change.

In CSPF, we don't have to recalculate permissible functions even after any transformation of the circuit within the permissible functions, while some don't-cares are missed. All gates and PIs in the circuit are ranked based on some heuristics before calculation of CSPF. Let R_i be the rank for a gate or PI *i*. For a PI pattern *V* where $G_i(V)$ is 1, $G_j(V)$ is kept at 0 where a gate or PI *j* is in FI_i, $F_j(V)$ is 0, and R_j is larger than R_{j0} for any other gate or PI *j0* in FI_i where $F_{j0}(V)$ is 0. For example, Figure 3 shows MSPF and CSPF for the same connections. All connections are made to be * in MSPF, on the contrary the third connection is kept at 0 in CSPF.

Table 2: The truth table of AND'.

AND'	0	1	*
0	0	0	0
1	0	1	1
*	0	1	*

Table 3: The truth table of @.

@		X	
		0	1
G_i	0	INVALID	1
	1	*	0
	*	*	*

Table 4: The truth table of #

#		F_{j0}	
		0	1
Y	0	0	*
	1	INVALID	1
	*	*	*

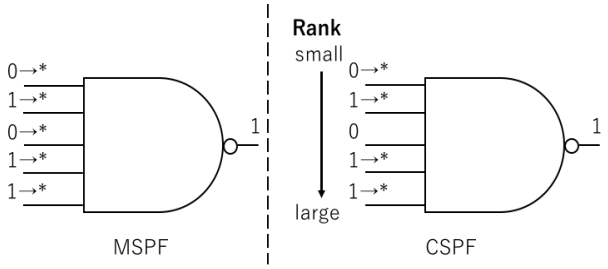

Figure 3: The difference between MSPF and CSPF.

CSPF is calculated in the following way. 1' and 2' are the same as 1 and 2 in the calculation of MSPF, while 3'-6' replace 3-5.

1'. For each gate *i* in a topological order from POs to PIs, do 2' and 3'.
2'. Calculate AND' (Table 2) of $G_{i,j}$ for every gate or PO *j* in FO_i, and the result is assigned to G_i.
3'. For each gate or PI *j0* in FI_i, do 4'-6'.
4'. Calculate AND of F_{j1} for every gate or PI *j1* in FI_i where R_{j1} is larger than R_{j0}. Let X be the result. (If there is no such *j1*, X is constant 1).
5'. Calculate @ (Table 3) of G_i and X. Let Y be the result.
6'. Calculate # (Table 4) of Y and F_{j0}, and the result is assigned to $G_{j0,i}$.

In 4', the range of *j1* is restricted to the input connections from gates or PIs with larger rank than the rank of the gate or PI *j0*. Thus, the input connection whose source has the largest rank in the sources of the input connections whose values are

0 is not treated as don't-care. If the output of the gate is 1 and an input connection is 1, the input connection is don't-care because there must be another input connection whose value is 0. 6' is based on this. Note the condition that the output is 0 and the input connection is 0 never happens.

3.2. Implementation of CSPF

The transduction method changes the structure of the circuit within permissible functions in order to reduce the number of gates and wires. Because functions for gates and connections are changed many times and the recalculation of MSPF is time-consuming, we decided to use CSPF.

A logic function in the circuit is represented by a BDD. Different from equivalence checking, we need all logic functions in the circuit not only logic functions at POs. These BDDs are built before the calculation of CSPF.

While calculating CSPF, we build another BDD representing * for each permissible function. Let G_DC_i and $G\_DC_{i,j}$ be the BDDs representing * in G_i and $G_{i,j}$ respectively. For any PI pattern *V* where $G_i(V)$ is *, $G\_DC_i(V)$ is 1. For any other PI pattern *Vn*, $G\_DC_i(Vn)$ is 0. The same relation holds between $G_{i,j}$ and $G\_DC_{i,j}$.

Code 1 is a pseudo code of the calculation of CSPF. CSPF() corresponds to 1', CSPF-gate(gate *i*) corresponds to 2', and CSPF-fanin(gate *i*) corresponds to 3'-6' in the calculation of CSPF explained in the previous section. The operations AND', @, and # are decomposed into normal logic operations with G_DC_i or $G\_DC_{i,j}$.

CSPF() removes redundant gates and wires while calculating CSPF. A wire whose permissible function consists of only 1 and * is redundant because if it is set to always 1, it never affects the output of the gate. If all input connections of a gate are removed, the output of the gate is replaced by constant 0 and the gate is removed. The gates without fanouts are also removed. At the end of CSPF(), logic functions for all gates are updated to reflect the removal of redundant gates and wires. Note the logic functions at the PIs/POs don't change.

The rank is calculated before CSPF() and is not updated during CSPF(). A connection from a gate or PI with smaller rank has more chances to have * in its permissible function. To reduce the number of gates, the rank of gate harder to be removed is set larger as follows. These are written in the descending order of priority.

1. The rank of a PI is larger than the rank of any gate.
2. The rank of a gate with larger number of fanout is larger.
3. The rank of a gate whose output has a logic function output 0 more frequently is larger.
4. The rank becomes larger in a topological order from POs to PIs.

3.3. The transduction method

Just calculating CSPF can reduce the number of gates and wires, but the transduction method transforms the circuit within permissible functions to reduce more.

We implemented 2 methods that allow us to rewire the network. The basic idea is to add the output of a gate or PI *j* to the input of another gate *i* where the connectable condition

```
CSPF-fanin( gate i ) {
  for each gate or PI j0 in FI_i {
    A:=constant 1;
    for each gate or PI j1 in FI_i where R_j1 > R_j0
       { A:=AND(F_j1, A); }
    B:=AND(F_i, NOT(A)); // @
    C:=OR(G_DC_i, B); // @
    D:=AND(F_i, F_j0); // #
    G_DC_j0,i:=OR(C, D); // #
    if OR(F_j0, G_DC_j0,i) is constant 1 { Remove c_j0,i; }
  }
  if FI_i is empty { Replace gate i by constant 0; }
}
CSPF-gate( gate i ) {
  if FO_i is empty { Remove gate i; return; }
  A:=constant 1;
  for each gate or PO j in FO_i
    { A:=AND(G_DC_i,j, A); } // AND'
  G_DC_i:=A;
}
CSPF() {
  For each gate i in a topological order from POs to
PIs { CSPF-gate( i ); CSPF-fanin( i ); }
  For each gate i in a topological order from PIs to
POs { Update F_i; }
}
```

Code 1: A pseudo code of the calculation of CSPF.

(1) is satisfied and to update permissible functions. The connectable condition is that F_i is included in G_i even after adding a new connection $c_{j,i}$. For a PI pattern *V* where $F_i(V)$ is 1, $F_i(V)$ doesn't change by any new $c_{j,i}$. For any other PI pattern *Vn*, $F_i(Vn)$ changes by a new $c_{j,i}$ only if $F_j(Vn)$ is 0. Therefore, a new $c_{j,i}$ can be created when the condition $NOR(F_i(V), F_j(V)) \rightarrow G\_DC_i(V)$ holds for any PI pattern *V*, and this is formulated as (1).

$$OR(F_i, F_j, G\_DC_i) = \text{constant } 1 \quad (1)$$

The rank is not always recalculated after the transformation of the circuit to save the processing time. It is updated only when we call Eager-CSPF, which updates the rank for all gates and calls CSPF() iteratively until there is no reduction in the number of gates and wires.

As the flow of optimization, we build BDDs for all gates, call Eager-CSPF, and apply the transduction method. Note redundant gates and wires are removed while updating CSPF.

The pseudo code of our first transduction method "Weak-reduce" is shown in Code 2. It tries all combinations of gates *i* and *j* in an exhaustive way, while CSPF is updated for the input connections of the gate *i* when the gate or PI *j* is added to its input. If no wire is removed in updating CSPF, the added input connection from the gate *j* is removed.

The second method "Eager-reduce" can be found in Code 3 with modification from Code 2 shown in red. When adding a connection from gate or PI *j* to gate *i*, some wires in fanin cone of gate *i* may become redundant and this method removes them. Moreover, it calls Eager-CSPF after each transformation of circuit.

```
Weak-reduce() {
  for each gate i in a topological order from POs to PIs
{
    for each gate or PI j in a topological order from PIs
to POs {
      if (j is i) or (j is in FIi) or (j is in FOCi)
        { continue; }
      if OR(Fi, Fj, G_DCi) is constant 1 {
        Create cj,i;
        CSPF-fanin( i );
        if the number of gates and wires is not reduced,
          { remove cj,i; }
        Update Fi;
    }}
    CSPF-fanin( i );
    for each gate j in FICi in a topological order from
POs to PIs { CSPF-gate( j ); CSPF-fanin( j ); }
    for each gate j in a topological order from PIs to
POs { Update Fj; }
}
```

Code 2: The first transduction method: Weak-reduce.

```
Eager-reduce() {
  for each gate i in a topological order from POs to PIs
{
    for each gate or PI j in a topological order from PIs
to POs {
      if (j is i) or (j is in FIi) or (j is in FOCi)
        { continue; }
      if OR(Fi, Fj, G_DCi) is constant 1 {
        Create cj,i;
        CSPF-fanin( i );
        for each gate k in FICi in a topological order
from POs to PIs { CSPF-gate( k ); CSPF-fanin( k ); }
        if the number of gates and wires is not reduced {
          remove cj,i;
          Update Fi; // necessary in case of Refresh
          for each gate k in FOCi in a topological order
from PIs to POs { Update Fk; }
          CSPF-fanin( i );
          for each gate k in FICi in a topological order
from POs to PIs { CSPF-gate( k ); CSPF-fanin( k ); }
        } else {
          for each gate k in a topological order from PIs
to POs { Update Fk; }
          Call Eager-CSPF;
        }
}}}
```

Code 3: The second transduction method : Eager-reduce.

When the method runs out of BDD nodes during its procedure, it removes all BDDs, builds BDDs for the current circuit, and calls CSPF(). This is called Refresh. If Refresh fails to build the BDDs or calculate CSPF due to the shortage of BDD nodes, more nodes are allocated, or the optimization is terminated.

4. Experimental setup

We conducted 3 experiments. In the first and second experiments, we compared the simple BDD package against CUDD [4]. The memory and the runtime were measured. To measure significant memory usage, we subtracted from the measured value the memory required to store the BDD of constant 0 function: 9 MB for the simple BDD package and 30 MB for CUDD.

The targets of first experiment were BDDs of combinational array multipliers from 8×8 to 18×18 bits generated by ABC. It is known that the number of BDD nodes needed to represent an integer multiplier is the smallest when the order of the variables is in the order of a4, b4, a3, b3, a2, b2, a1, b1 on 4×4 bits multiplier for example. This order was used in our experiments. Note that both our implementation and CUDD cannot build BDD of the 20×20 bits multiplier because the number of BDD nodes exceeds 2^{31}.

CUDD allows the user to disable dynamic variable reordering, and we measured the performance of CUDD without dynamic reordering in the first experiment. Our program requires the initial number of allocated BDD nodes as a power of 2 ($nAlloc=2^n$) and we measured its performance with 3 different values of n: the minimum n where 2^n is more than the number of PIs (the reallocation will be performed when building BDD), the minimum n where the BDDs of each multiplier are built without reallocation, and 31 (2^{31} BDD nodes) which is the upper limit of the program.

The second experiment was a comparison against CUDD with dynamic variable reordering. We experimented on multi-level combinational circuits with 20-200 inputs and less than 2000 gates in LGSynth91 benchmark [6]. We measured how the number of BDD nodes affects memory usage and runtime.

The third experiment was a performance comparison of the network optimization methods. Our implementation was compared with ABC [5]. We compared the following 4 method: an ordinary optimization by ABC, a script of ABC optimization commands which uses don't-cares in terms of PIs, the our first transduction method "Weak-reduce", and the combination of "Weak-reduce" and the second one "Eager-reduce".

The ordinary optimization by ABC uses commands "dc2", "resyn", "resyn2", and "resyn3". We repeatedly applied these commands in the following policy: 1. Apply “dc2” 10 times. If the number of AIG nodes is reduced, do 1 again. 2. If the number of AIG nodes is no longer reduced even if dc2 is applied 10 times, apply "resyn", 10 times of "dc2", "resyn2" 10 times of "dc2", "resyn3", and 10 times of "dc2". If the number of AIG nodes is reduced after those commands, return to 1. Otherwise, finish the optimization.

Another ABC optimization uses a script "if -K 6 -m; mfs2 -W 1000000 -F 1000000 -D 1000000 -L 1000000 -C 1000000 -e; strash; compress2rs". We applied this script 300 times and measured the minimum number of AIG nodes. Note that this command doesn't cause monotonic decrease in the number of nodes. We report only the runtime needed to reach the network with the minimum number of AIG nodes.

Command "if" is a simple technology mapper, which transforms AIG into a network of logic nodes. With option,

Table 5: The memory usage and runtime to build BDDs of multipliers using the simple BDD package and CUDD without dynamic variable reordering. Both package reallocated BDD nodes during the computation incrementally.

Multiplier			The simple BDD		CUDD	
Bit	AIG nodes	BDD nodes	Mem (MB)	Runtime (sec)	Mem (MB)	Runtime (sec)
8×8	424	19,830	1	0.02	4	0.02
10×10	690	184,449	9	0.18	44	0.34
12×12	1,020	1,709,060	76	2.92	160	6.83
14×14	1,414	15,877,043	502	43.71	1426	84.64
16×16	1,872	147,590,995	7722	478.29	13266	1071.96
18×18	2,394	1,374,416,471	61441	4991.34	117346	13682.34
Ratio			0.40	0.55	1.00	1.00

Table 6: The memory usage and runtime to build BDDs of multipliers using the simple BDD package. The simple BDD package initially allocates 2^{15}, 2^{18}, 2^{21}, 2^{24}, 2^{28}, and 2^{31} BDD nodes for the minimum number of BDD nodes without reallocation.

Multiplier	The minimum number of BDD nodes without reallocation		2^{31} BDD nodes allocated initially	
Bit	Mem (MB)	Runtime (sec)	Mem (MB)	Runtime (sec)
8×8	0	0.01	32749	7.92
10×10	7	0.13	32775	8.31
12×12	59	2.14	32860	10.23
14×14	479	35.57	33634	37.98
16×16	7679	303.27	41072	222.21
18×18	61430	3323.45	61430	3323.45
Ratio	0.32	0.36	1520.99	70.47

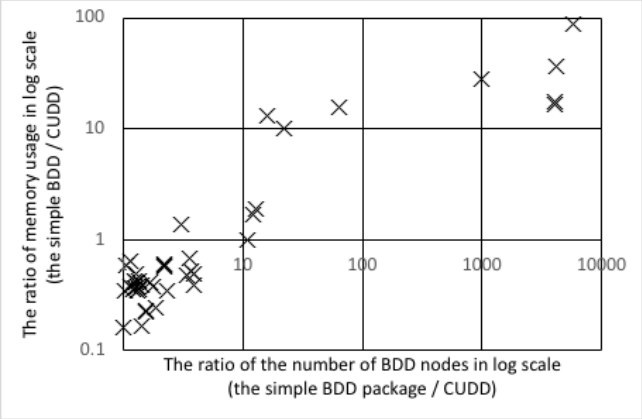

Figure 4: The ratio of memory and the number of BDD nodes used in the simple BDD package over CUDD with dynamic variable reordering.

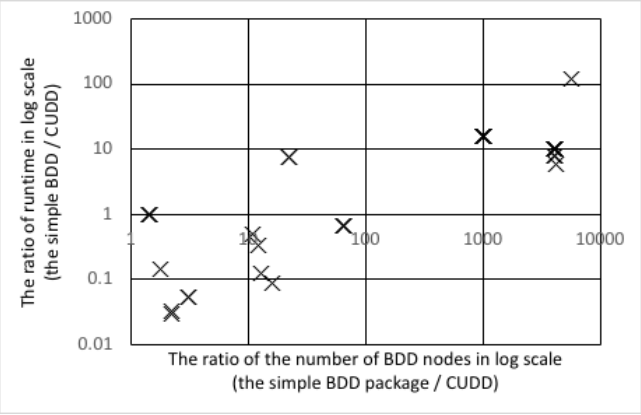

Figure 5: The ratio of runtime and the number of BDD nodes used in the simple BDD package over CUDD with dynamic variable reordering.

"-K 6" it maps AIG into 6-input LUT network while option "-m" enables cut minimization by removing vacuous variables. Command "mfs2" is the SAT-based optimization used with options to make sure it works on a complete circuit (and not just on a window). It is a good match with CSPF-based optimization, which uses BDDs in terms of PIs. Option "-e" enables high-effort resubstitution. Finally, "compress2rs" is an old AIG optimization script, which performs rewriting, refactoring, balancing, and truth-table-based resubstitution.

The third method using "Weak-reduce" was executed as follows: First, we calculate the functions in the whole circuit and call Eager-CSPF. Next, we repeatedly apply "Weak-reduce" until there is no improvement. Then we change the structure of the circuit by "strash". Again, we read the circuit, calculate the function, perform Eager-CSPF, and apply "Weak-reduce" repeatedly. We did this iteratively until there is no reduction. Our program allocated 2^{21} BDD nodes initially and reallocated 2^{31} BDD nodes when it runs out of BDD nodes during Refresh. The optimization was terminated when it required more than 2^{31} BDD nodes (out of memory) or took more than 1 hour (out of time).

The last method, the combination of "Weak-reduce" and "Eager-reduce", was performed by doing the procedure explained above for "Weak-reduce" and the same for "Eager-reduce". This was also repeated until there was no reduction.

The experiment was conducted on the circuits synthesized from PLA (Programmable Logic Array) format [7] benchmarks in LGSynth91. The synthesis from PLA was

performed by "fx", which is the traditional fast_extract algorithm used to transform PLA into a multi-level circuit [8].

Ratio in the tables was calculated by dividing a value for each benchmark and averaging the quotients in all benchmarks.

All experiments were done by single thread in the following environment:

- CPU: Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
- Memory: 512 GB
- OS: OpenSUSE Tumbleweed 20190327

5. Experimental results

The result of the first experiment is shown in Table 5 and 6. In Table 5, the simple BDD package initially allocated 2^5 BDD nodes for 8×8 to 14×14, and 2^6 BDD nodes for 16×16 and 18×18 bit multipliers and reallocated doubled number of BDD nodes incrementally up to 2^{15}, 2^{18}, 2^{21}, 2^{24}, 2^{28}, and 2^{31} BDD nodes for 8×8, 10×10, 12×12, 14×14, 16×16, and 18×18 bit multiplier respectively. On the other hand, in Table 6, the simple BDD package allocated those numbers of BDD nodes or 2^{31} BDD nodes initially and finished building BDDs without reallocation.

The simple BDD package was around 2 times faster than CUDD and used only 60% memory compared to CUDD even if it does the reallocation. The reallocation took around 30% of time compared to the results of the simple BDD package without reallocation. For 14×14 bits and larger multipliers, to initially allocate 2^{31} BDD nodes was the fastest in the results. A good way to use the new package is to start from the small number of BDD nodes with incremental reallocation and to jump to 2^{31} BDD nodes if it requires more than 2^{21} BDD nodes, while the numbers depend on the machine.

Figure 4 and 5 show the rate of memory and runtime increased by the absence of dynamic variable reordering in the second experiment in double logarithmic graph. Figure 5 shows only the points where both packages took more than 0 second. The ratio of memory was linearly increased on the ratio of the number of BDD nodes, and CUDD with dynamic reordering used smaller size of memory than the simple BDD package when the ratio of BDD nodes was more than 10x. The runtime also increased in the same way while CUDD takes time for reordering.

Table 7 is the results of the third experiment which is comparing the results of optimization. Our method (combination of "Weak-reduce" and "Eager-reduce") reduced the number of AIG nodes 20% more than ordinary ABC optimization and 10% more than the ABC script using "mfs2" on average.

As is shown in [9], these methods using don't-care in terms of PIs were very effective for *alu4*, where AIG node count is around 20 times smaller than the original.

6. Conclusion

We implemented a simple BDD package without variable reordering and used it in the optimization program based on the transduction method.

When evaluating the BDD package alone, it was found that memory is reduced by 40% and the runtime is 2x2 faster, compared to CUDD without dynamic variable reordering if the good variable ordering is given. We also found that memory and runtime improvements are expected against CUDD with dynamic reordering if the increase of BDD nodes without dynamic reordering is less than 10x.

When experimenting on the optimization methods, our implementation was possible to achieve substantial AIG node reduction (about 10-20%) compared to ABC although it was 50x slower.

The program is available at [10].

Acknowledgements

This work was supported in part by SRC Contracts 2710.001 and 2867.001.

References

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," in *IEEE Transactions on Computers*, vol. C-35(8), pp. 677-691, Aug. 1986.

[2] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, vol. 40(2), pp. 205-213, Feb. 1991.

[3] S. Muroga, Y. Kambayashi, H. C. Lai and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," in *IEEE Transactions on Computers*, vol. 38(10), pp. 1404-1424, Oct. 1989.

[4] F. Somenzi. CUDD Package, http://vlsi.colorado.edu/~fabio/

[5] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, https://github.com/berkeley-abc/abc

[6] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," in *Technical Report 1991-IWLS-UG-Saeyang*, MCNC.

[7] PLA Format, http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/espresso.5.html

[8] J. Rajski, J. Vasudevamurthy, "The test-preserving concurrent decomposition and factorization of Boolean expressions", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11 (6), pp.778-793, June 1992.

[9] Y. Matsunaga and M. Fujita, "Multi-level logic optimization using binary decision diagrams," in *Proceedings of International Conference on Computer Aided Design*, 1989.

[10] https://github.com/MyskYko/abc

Table 7: The number of AIG nodes and runtime in a comparison of the optimization methods. ABC1 is the ordinary optimization by ABC. ABC2 is the ABC optimization with a script using if, mfs2, and compress2rs. Weak uses the first our implementation of transduction method "Weak-reduce". Combo uses both of our implementations "Weak-reduce" and "Eager-reduce". MO or TO means out of memory or time respectively. Ratio is calculated excluding benchmarks where MO or TO happened. The minimum number of AIG nodes is highlighted in red for each benchmark with reduction.

testcase	AIG nodes	Optimized AIG nodes				Runtime (sec)			
		ABC1	ABC2	Weak	Combo	ABC1	ABC2	Weak	Combo
xor5	12	12	12	12	12	0.09	0.00	0.00	0.00
con1	17	16	17	16	16	0.14	0.00	0.00	0.01
t481	25	25	25	25	25	0.15	0.00	0.00	0.01
rd53	36	30	26	29	24	0.14	0.04	0.00	0.03
misex1	50	48	45	34	34	0.19	0.05	0.02	0.06
squar5	56	41	44	50	49	0.34	0.05	0.02	0.08
cordic	65	53	51	54	54	0.27	0.03	0.04	0.68
vg2	73	73	70	70	68	0.19	0.11	0.19	1.75
b12	74	51	52	63	60	0.36	0.13	0.03	0.24
misex2	74	74	74	71	71	0.18	0.00	0.06	0.26
rd73	94	63	72	70	61	0.25	0.35	0.09	0.31
5xp1	98	72	62	74	58	0.44	0.22	0.03	0.28
inc	113	100	107	95	86	0.62	0.07	0.12	0.59
sao2	121	111	104	107	102	0.32	0.20	0.18	1.44
e64	127	127	127	127	127	0.20	0.00	0.27	5.18
o64	129	129	129	MO	MO	0.24	0.00	-	-
clip	137	110	59	85	75	0.55	0.38	0.14	0.56
bw	142	116	118	117	105	0.75	0.09	0.17	3.13
Z9sym	146	142	130	92	79	0.71	1.37	0.17	1.25
Z5xp1	154	82	54	105	88	0.73	0.25	0.14	1.21
rd84	159	126	120	88	63	0.92	0.91	0.22	0.49
9sym	202	185	158	152	124	0.82	3.93	0.98	13.00
duke2	283	256	257	240	225	0.94	2.81	2.38	36.36
apex2	289	183	145	128	91	1.44	1.12	217.08	993.16
ex4	407	337	378	343	326	1.15	0.10	88.36	201.52
misex3c	465	412	354	332	310	0.90	4.57	8.14	131.55
table5	558	489	530	476	447	3.26	7.07	15.04	394.09
apex5	665	618	601	TO	TO	2.50	0.32	-	-
spla	671	533	258	258	233	2.89	1.23	3.67	33.19
table3	696	611	651	584	532	7.75	48.69	23.87	985.63
ex5	769	351	227	273	199	4.30	1.90	5.57	94.33
misex3	841	607	474	325	288	7.72	5.57	27.20	242.39
cps	878	745	734	600	502	4.73	11.06	65.53	2234.62
apex1	886	795	840	TO	TO	4.33	7.26	-	-
alu4	1052	851	68	97	83	9.65	5.21	26.21	25.35
apex3	1114	1066	1114	MO	MO	7.32	0.00	-	-
pdc	1319	1107	278	267	234	17.79	1.63	6.80	44.72
seq	1378	1133	930	736	TO	11.99	119.49	3363.57	-
apex4	1743	1692	1743	1680	TO	18.69	0.00	549.87	-
ex1010	1747	1621	1747	1712	TO	37.67	0.00	986.74	-
Ratio	1.00	0.84	0.74	0.72	0.66	1.00	1.07	8.38	57.98