Generalized Buffering of PTL Logic Stages using Boolean Division and Don't Cares

Rajesh Garg Sunil P Khatri
Department of ECE, Texas A&M University, College Station, TX 77843

Abstract— **Pass Transistor Logic (PTL) is a well known approach for implementing digital circuits. In order to handle larger designs, and also to ensure that the total number of series devices in the resulting circuit is bounded, partitioned Reduced Ordered Binary Decision Diagrams (ROBDDs) can be used to generate the PTL circuit. The output signals of each partitioned block typically needs to be buffered. In this paper, we present a methodology to perform *generalized* buffering of the outputs of PTL blocks using Boolean division and *compatible observability don't cares* (CODCs). By performing the Boolean division of each PTL block using different gates in a library, we select the gate that results in the largest reduction in the depth of the PTL block. The use of CODCs further simplifies the logic of the PTL block. The gates serve the function of buffering the outputs of the PTL blocks, while also reducing the depth and delay of the PTL block. Since the CODC computation is memory intensive and time consuming, CODCs can not be computed for large circuits. To avert this problem we also use approximate CODCs (ACODCs) which can be computed for arbitrary size circuit. Over a number of examples, we demonstrate that our approach (generalized buffering with ACODCs) results in a 29% reduction in circuit delay and 27% reduction in number of MUXes required, with an improvement of 5% in circuit area, compared to a traditional buffered PTL implementation of the circuit. Our approach also resulted in a delay reduction of 5% and an area reduction of 2% over generalized buffering without don't cares.**

I. Introduction

Pass Transistor Logic (PTL) is a circuit implementation style that has typically been used for many specific circuit implementations, like barrel shifters. Although PTL offers great benefits for such designs, there has been no widely accepted PTL design methodology that could be used to make PTL more broadly acceptable. There have however been several efforts at the research level, to explore the promise of PTL.

Synthesis approaches for PTL structures typically leverage the fact that there is a direct mapping between the ROBDD [1], [2] and the PTL implementation of a circuit. In fact, each ROBDD node can be mapped to a MUX (which can be implemented using NMOS or CMOS devices). Figure 1 illustrates the mapping between an ROBDD and PTL structure. In Figure 1 the solid line (dashed line) represents the positive (negative) cofactor of an ROBDD node with respect to the variable of that node. For the PTL implementation of the ROBDD of any function, there is an isomorphism between the connectivity of the ROBDD nodes and the MUXes in the PTL implementation. This elegant and easy mapping between ROBDDs and PTL structures is not without attendant problems, however.

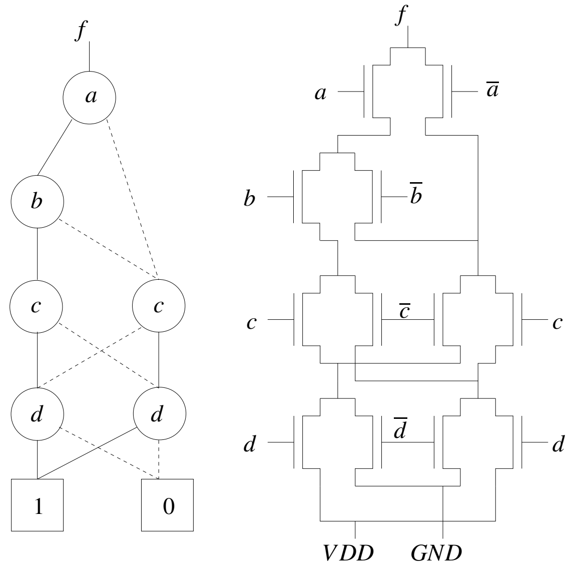


Fig. 1. ROBDD and its PTL Implementation

- For one, in a bulk MOS implementation of any circuit, it is not practical to connect more than 4-5 devices in series, due to the phenomenon called *body effect*, which effectively increases the threshold voltage V_T of most of the series-connected MOSFETs. This results in a slower design. Body effect is governed by the equation
  $V_T = V_T^0 + \gamma\sqrt{V_{sb}}$
  Here V_T is the threshold voltage of the device, V_T^0 is the threshold voltage of the device at zero body bias, γ is the body effect coefficient, and V_{sb} is the source to bulk voltage of the MOSFET. When several MOSFETs are connected in series in a PTL circuit, all but one of them is affected by body effect. For this reason, in practice, VLSI designers do not stack more than 4-5 devices in series.
  This results in a problem for the traditional PTL implementations, which attempted to build monolithic ROBDDs. To solve this problem, we need to build ROBDDs in a partitioned [3] manner, such that if an intermediate ROBDD has a depth greater than 4 or 5, a new variable is created. In circuit terms, a buffer is implemented at the output of the new variable, ensuring that the circuit drive capability is regenerated every few

levels in the final PTL design.
- The second problem is that ROBDDs, if built monolithically, can exhibit unpredictable memory explosion. This precludes the applicability of the PTL methodology for large designs, *as long as the ROBDDs are build monolithically* (since it is very hard to build monolithic ROBDDs of large functions). Again, this suggests the use of partitioned ROBDDs to averts this problem.

In summary, partitioned ROBDD construction helps tackle both the above problems associated with PTL synthesis. In such a partitioned PTL design, the outputs of each PTL structure is buffered, to regenerate the electrical drive capability every 4 or 5 levels.

This work describes a new PTL synthesis approach which performs better than buffered partitioned ROBDD based PTL synthesis. In principle, it still employs partitioned ROBDDs, however, the buffering between PTL stages is done using *generalized buffers*. These could be arbitrary gates in the cell library. We achieve this by casting the problem of generalized buffering as an instance of Boolean division. *Compatible observability don't cares (CODCs)* are used along with Boolean division. By using more complex gates for the buffering logic, and by additionally using CODCs, our method is able to significantly simplify the logic in the PTL structure, resulting in an improvement (about 29% on average) in total circuit delay, which is achieved by significantly reducing the total number of MUXes. Even after accounting for the increased area of the generalized buffers, our method comes out slightly ahead in terms of area (by about 5% on average). The above comparisons are against traditionally buffered partitioned ROBDD-based PTL structures.

The don't cares (CODCs) can be computed using *full_simplify* in SIS [4]. However, this computation is very memory intensive and time consuming because of its use of ROBDDs. As a result, the CODC computation is typically not feasible for large circuits. The work presented in this paper is applicable for large circuits as well, since it uses an approximation of CODCs, which can be computed efficiently and in a robust manner. In [5], the authors presented a technique to compute approximate compatible don't cares (ACODCs). These don't cares are an approximation of CODCs. The computation of the ACODC of any node n is based on extracting other nodes in the transitive fanout/fanin (TFO/TFI) of n, up to a limited depth. The resulting nodes induce a sub-network of the original network. The ACODC of n is simply its CODCs computed for the induced sub-network. ACODCs can be computed for arbitrarily large designs, and the time and memory utilization for the ACODC computation is much less (both typically 30X lower) than the corresponding values for a full CODC computation. At the same time, the literal count reduction obtained by ACODCs is typically about 80% of that obtained by full CODCs. Hence, ACODCs are used in this paper, allowing the applicability of the proposed approach to industrial designs.

The remainder of this paper is organized as follows. Section II discusses some previous work in this area. In Section III we describe our method of generalized buffering of PTL structures, based on Boolean division. In Section IV we present experimental results comparing our idea with traditional buffered partitioned ROBDD-based PTL designs [6]. We have also compared our approach with generalized buffering without CODCs [7]. Conclusions and future work are discussed in Section V.

II. Previous Work

There has been a significant amount of work in the area of pass transistor logic synthesis. In addition, there have been several reported design variations on the PTL circuit concept. A good reference source for published work in this area is [8]. The focus of our paper being logic synthesis for PTL design, we will not discuss circuit-level variations of the PTL approach. Given the generality of our synthesis methodology, it is orthogonal to the circuit issues and ideas that are discussed in the papers referred to in [8].

In [9], Yano et al. present a synthesis tool, which works with their PTL library of cells, macrocells etc. The philosophy of Macchiarulo et al. [10] is that while PTL based synthesis is well researched, layout and back-end tools for PTL are not found as readily. They describe a layout generator, to help develop more complete tools for PTL design. The approach of Radhakrishnan et al. in [11] is motivated by the need to develop high density, low power, high performance circuits using PTL. The authors report a synthesis method, using two approaches – a modified Karnaugh map [12] approach for small circuits, and a Quine-McCluskey [13], [14] based approach for larger designs. In [15], the authors study regenerative pass transistor logic (RPL), a dual rail PTL family. Their interest stems from the compact area of these circuits. In [16], Hsiao et al. present a layout and logic synthesis approach, with the input being a logic function, while the output is a PTL netlist, using MUXes and inverters. After synthesizing the logic, the methodology adds inverters, in order to ensure that the longest series path of MUXes has a bounded depth. In this approach, the buffering is simple (using inverters) as opposed to the generalized buffering (using arbitrary library gates) in our work. The work of [6] builds ROBDDs [1], [2] in a partitioned manner [3], thereby avoiding the memory blow-up that often occurs while using ROBDDs. The resulting small ROBDDs are directly mapped to PTL structures, resulting in an efficient synthesis methodology. The downside of this approach is once again, the absence of generalized buffering. This approach also does not utilizes the CODCs to simplify the logic of PTL structures. The authors allude to performing PTL synthesis in a manner that maps some nodes to CMOS gates while others are mapped into PTL blocks. However, this leaves the designer little control of the depth of these sections. In contrast, our generalized buffering approach guarantees a fixed bound on the depth of the PTL block, and also ensures just one level of generalized buffers between PTL blocks. Our generalized buffering approach also uses CODCs to simplify the PTL blocks.

In [17], an approach using multilevel gates along with PTL and transmission gates is reported by Neves et al., producing a regular and dense layout. This approach permits buffer inclusion. It can be viewed as an approach that combines synthesis and layout in the PTL design flow, and is as such orthogonal to our approach. The work of Pedron et al. in the paper [18] reports synthesis algorithms for PTL. Minimization is performed by using incomplete transmission gates. The results of synthesis using their tool PAVOS demonstrate good quality results when compared with manually tuned PTL circuits. The work of Markovic et al. [19] reports on PTL synthesis for Complementary Pass Transistor Logic (CPL), Dual Pass Transistor Logic (DPL) and Dual Voltage Logic (DVL). Finally, the work of Shelar et al. in the paper [20] reports on a novel method to minimize power in a PTL structure, by transforming the problem into one of ROBDD decomposition and solving it with a max-flow min-cut approach.

In [21], a mixed PTL-CMOS approach was presented by Lai et al. The decomposition process simply extracted unate variables from a sum-of-products (SOP) representation of a function and recursively cofactored these variables (creating MUXes in the process) until unate leaves were reached. Unate leaves were realized using library gates alone. The weakness of this approach is that it starts with a SOP representation, limiting its effectiveness for large designs. Further, if a function is unate, then the entire circuit realization is purely standard-cell based. Our approach, in contrast, can handle arbitrarily large designs since it starts with a partitioned ROBDD construction as the first step. Also, it works equally effectively for unate designs.

Another mixed PTL-CMOS approach was presented by Yamashitas et al. in the paper [22]. In this paper, the circuit structure was a AND gate followed by a MUX. In contrast to our approach, this scheme did not selectively divide a library of standard cells into the PTL structure, thereby not exploiting the flexibility available in the Boolean division process. In [23], the authors present a PTL synthesis algorithm with generalized buffering. They cast the problem of generalized buffering as an instance of Boolean division. Their approach yields a good reduction of total circuit delay over traditional buffering. They also obtained a small improvement in total circuit area over traditional buffering. However, their approach does not utilizes the CODCs to simplify the PTL blocks.

In our approach, we also perform generalized buffering using Boolean division. However, multi-level don't cares (CODCs & ACODCs) are used during Boolean division, to further simplify the PTL structure. The simplification of PTL structures will results in a further reduction of total circuit delay and area.

III. Our Approach

We define the *depth* of an ROBDD g as the maximum number of nodes in any traversal of g to its terminal nodes. Our method creates partitioned PTL structures with a maximum depth of 5 (this number can be arbitrary, though). This ensures that there are no more than 5 transistors in series, in any PTL block.

In order to implement generalized buffering, we employ Boolean division with CODCs. Algorithm 1 describes the new PTL synthesis methodology which uses generalized buffers and CODCs to simplify the PTL blocks. Consider a boolean network η. First the network η is optimized and decomposed using 2-input gates and inverters only. This is done to ensure that the PTL structure grows in predictable manner. Let the new network be referred to as η^*. Now the nodes of η^* is sorted in depth-first manner. The resulting array of nodes is sorted in *levelization* order, and placed into an array A.

In this approach, we first build ROBDDs of the nodes of η, topologically from the inputs to the outputs by fetching the nodes from the array A in index order. Suppose we encounter a new node n for which we want to construct a ROBDD f. The ROBDD of each of its fanins must have a support of at most 4 variables (assuming that the fanins are not partitioned ROBDD variables). When constructing the ROBDD f of n, it can initially have at most 8 ROBDD variables (since any node in the original network can have at most 2 fanins). In case no division is possible, we can make one of the inputs of n a new variable, yielding an ROBDD of depth at most 5. If the depth of f is less than 5 then we continue with creating the ROBDD of the other nodes in array A. However, if the depth of f is greater than or equal to 5 then we compute the CODCs (or ACODCs) of the node using *compute_dc* function. Then we attempt to divide f with the gates in a library by calling the *test_division* function. The detailed description of the *test_division* function is provided after the synthesis flow. If the division is successful, then the depth of the ROBDD f will reduce. If the resulting depth of ROBDD is less than or equal to 5 than we made the node n a ROBDD variable otherwise we *back-track*. The *back-track* procedure is described later.

The key idea is that we will take the ROBDD of n, and attempt to divide it with the gates in a library. If such a Boolean division is possible *and* it is strictly depth-reducing, we select it. The test for whether a library gate g, with an associated variable G, divides the ROBDD f of n is described next. A pictorial view of the process is shown in Figure 2.

A. Boolean Division with Library Gates

Definition 1: g is a **Boolean divisor** of f if h and r exist such that $f = gh + r$, $gh \neq \emptyset$.

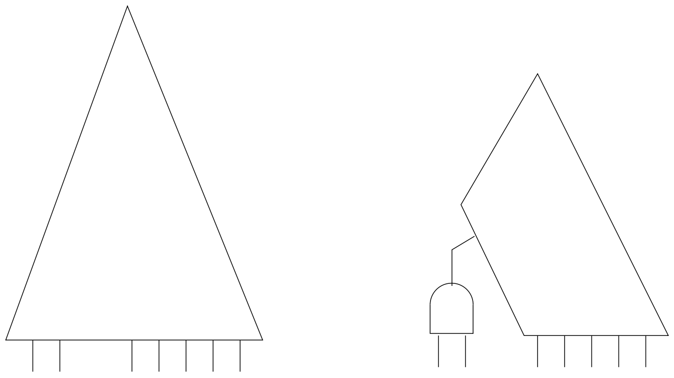
g is said to be a **Boolean factor** of f if, in addition, $r = \emptyset$, i.e., $f = gh$.

In this case, h is called the **quotient** and r is called the **remainder**. Note that h and r are not unique.

Theorem 3.1: If $fg \neq \emptyset$, then g is a Boolean divisor of f.

Algorithm 1 Pseudocode for PTL synthesis using generalized buffers and CODCs

```
η = optimize_network(η)
η* = decompose_network(η, p)
A = dfs_and_levelize_nodes(η*)
N* = 0
i = 1
while i ≤ size(A) do
  n = array_fetch(A,i)
  f = ntbdd_node_to_bdd(n)
  if bdd_depth(f) ≥ 5 then
    d = compute_dc(η*,n)
    for g ≡ G ∈ Gate Library do
      f = test_division(f,d,g,G)
    end for
    if bdd_depth(f) > 5 then
      back-track
    else
      bdd_create_variable(n)
      continue
    end if
  else
    continue
  end if
end while
```



a) Before dividing AND gate
b) After dividing AND gate (note depth reduced otherwise the gate is rejected

Fig. 2. Dividing a Generalized Buffer into a PTL Structure

Proof: If $fg \neq \emptyset$, we can write
$f = fg + f\overline{g}$
$f = g(f+x) + f\overline{g}$ where $x \subseteq \overline{g}$
which is of the form $f = gh + r$. ■

However, if f is an Incompletely Specified Function (ISF), with don't care d, then
$f = G(f+d+\overline{g}) + (f+d)\overline{g}$.

Therefore, the upper and lower bounds (U and L) for f are:
$U = G(f+d+\overline{g}) + (f+d)\overline{g}$
and
$L = (fG + f\overline{g})$

As a consequence, the test that we perform when we want to try to divide a gate g (having a variable G) into the ROBDD f is shown in Algorithm 2.

Algorithm 2 Pseudocode for Division of f by $g \equiv G$ using CODCs

```
test_division(f,d,g,G){
if $fg \neq 0$ then
  $L = (fG + f\overline{g})(g\overline{\oplus}G)$
  $U = (fG + dG + G\overline{g} + f\overline{g} + d\overline{g})(g\overline{\oplus}G)$
  $Z = bdd\_between(L,U)$
  $Z^* = bdd\_smooth(Z, gvars)$
  $R = bdd\_compose(Z^*, G, g)$
  if $f \subseteq R \subseteq f+d$ and $bdd\_depth(Z^*) < bdd\_depth(f)$
  then
    $Q = d(g\overline{\oplus}G)$
    $d = \exists_V Q$
    return(success, Z*)
  end if
else
  return fail
end if
}
```

In Algorithm 2, the functions L and U are ANDed with $(g\overline{\oplus}G)$ to express the fact that g and G are not independent variables, but rather are related as $G \equiv g$. Note that as a consequence of this, the $G\overline{g}$ term in line 4 of Algorithm 2 can be removed. We next find a small ROBDD Z (using the function *bdd_between*(), which returns the heuristically smallest ROBDD Z such that $L \subseteq Z \subseteq L+U$), which is a Boolean divisor of f. Next, we smooth out[1] the variables of g. If the resulting ROBDD Z^*, with g composed back into G, lies between f and $f+d$ and also if the depth of Z^* is less than the depth of f, we return Z^* as the quotient. If we have a valid division then we map the don't care d of the node n onto a new set of variables V so that it can be used during the next iteration. The CODCs are mapped using following equations:
$Q = d(g\overline{\oplus}G)$
$d = \exists_V Q$
where V is the set of variables which lie in the support of g but are not present in the support of Z^*. The re-mapped don't cares of n are used for further Boolean division, to explore additional generalized buffering opportunities for the node n.

B. Back-track

When we perform ROBDD construction, the maximum depth of the ROBDD of a node n before division can be 8, as discussed earlier. Our division routines systematically reduce this depth to below 5, by using several generalized buffers. If

[1]Smoothing is also referred to as Existential Quantification. The existential quantification of f with respect to a variable x is expressed as $\exists_x f = f_x + f_{\overline{x}}$. Smoothing a set of variables is achieved by performing existential quantification, one variable at a time

an ROBDD is not able to be divided (and the resulting depth after division is greater than 5), then we back-track, and make one of the fanins of n a new variable. The fanin which is made a new variable is the one whose topological level is one less than that of n. The reason for this is illustrated in Figure 3.

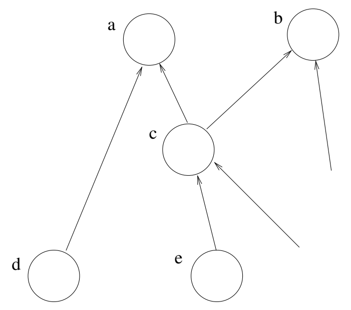


Fig. 3. Back-tracking during Division

Consider node a, at topological level n. Suppose it has two fanins c and d, with levels $n-1$ and $n-2$ respectively. Since partitioned ROBDD construction occurs in topological order from inputs to outputs, it is possible that node b has already been processed, and divided. Its ROBDD therefore has a depth less than 5 nodes. Now suppose the ROBDD depths of d and c are 4 each, and the ROBDD of a has depth 8. Suppose all divisions for node a fail. In that case, we need to back-track and make either c or d a new variable. This would guarantee that a has a new depth 5. We select c (with level $n-1$) as the new variable, since it is more likely that d (which is at a lower topological level) has more fanouts at level n or $n-1$. When c is made a new variable, all its fanouts are checked. If any of them has already been processed (such fanouts must have level n), then their ROBDDs are re-computed, and division is re-done. In this way, the PTL network is kept as small as possible. If this re-computation were not performed, then the logic of the node c would be implemented twice as a PTL structure (once for the node b and then again for the variable corresponding to node c itself).

If, after a back-track, the depth of the node n is less than 5, then we continue to grow the corresponding ROBDD further, up to a depth 8.

After attempting division (regardless of whether the division succeeded or failed), the depth of n is guaranteed to be less than or equal to 5, allowing for an elegant exit in case division fails. In general, however, this division strategy yields a number of good generalized buffers, so this occurrence is rare.

IV. Experimental Results

We implemented the new PTL synthesis algorithm with generalized buffering and don't cares in SIS [4]. Our method creates partitioned PTL structures with a maximum depth of 5 (this number can be arbitrary, though). This ensures that there are no more than 5 transistors in series, in any PTL block. The code consisted of reading a circuit (which was decomposed before-hand into 2-input gates and inverters), and then building ROBDDs of its nodes in a topological manner from the inputs to the outputs. When the depth of the ROBDD of any node grew beyond 5, division was invoked, as described in the previous section. The CODCs were computed and used during division. The CODC computation was done in two ways: one using $full\_simplify$ [4] (which provides complete CODCs) and another using approximate CODCs (ACODCs) as mentioned in [5]. Results are presented and compared for both styles of CODCs.

The resulting library gates that were utilized for division, and the new ROBDDs after division were stored within each node's data structure. Since the ROBDDs in question were small (with a maximum initial support of 8), division was performed exhaustively. If the resulting depth after division was still greater than 5, then a back-track step was invoked making one of the node's fanins a new variable. After this we check the nodes which are in the fanout of the fanin node corresponding to the newly created variable. If any of these fanout nodes was already processed, then their ROBDDs were re-computed, and division was re-done. After attempting division (regardless of whether the division succeeds or fails), the depth of the final ROBDD is guaranteed to be less than or equal to 5, allowing for an elegant exit strategy in case division fails. Our library consisted of the gates AND2, AND3, AND4, OR2, OR3 and OR4. Since any divided gate becomes a new variable, it is required in both its polarities. Therefore, by DeMorgan's law, we only need non-inverting gates in our library.

The MUX and all gates in the library were characterized for delay in SPICE [24], using a 100nm BPTM [25] process technology. Table I shows the delay and active area of MUX and all gates in the library.

Gate	Delay(ps)	Area(μ^2)
MUX	18	0.08
INV	10.26	0.08
Buffer	20.5	0.16
AND2	30.20	0.28
AND3	37.76	0.44
AND4	47.39	0.64
OR2	38.70	0.36
OR3	46.08	0.68
OR4	68.28	1.12

TABLE I
Delay of various gates in the library

The delay of a synthesized PTL circuit was extracted by finding the longest delay path from any output to any input. Table II compares the delay of the traditionally buffered partitioned ROBDD based implementation with the generalized buffering methodology (with and without CODCs). In this table, column 1 lists the example under consideration. Column 2 reports the circuit delay, for traditional buffering. The traditional method used for comparison was similar to that reported in [6]. Column 3 reports delay for PTL synthesis algorithm with generalized buffering without CODCs similar to that reported in [7] (as a fraction of the delay of the traditional method). Column 4 and 5 report delay number for new PTL synthesis using ACODCs and CODCs respectively (again as a fraction of the delay of the traditional method). Some entries in the table are marked as "-". This indicates that *full_simplify* was not able to compute the CODCs for that circuit. We observe that our method (i.e. generalized buffering with ACODCs) results in a speed-up of about 29% on average compared to the traditional method, and about 5% over generalized buffering without CODCs. The usage of complete CODCs using *full_simplify* yield a small improvement in delay of less than 1% in comparison with ACODCs.

	Traditional	Generalized Buffering		
	Buffering	Without CODC's	With ACODCs	With CODCs
Ckt	Delay (ps)	Delay	Delay	Delay
alu2	1927.26	0.53	0.55	0.54
alu4	3458.88	0.45	0.43	-
apex6	819.72	0.73	0.75	0.75
C432	2302.38	0.86	0.79	0.76
C499	670.68	1.01	1.01	1.01
C880	1248.8	0.70	0.53	0.54
C1908	1336.14	0.74	0.67	0.68
C3540	2104.56	0.71	0.53	-
C5315	1325.88	1.04	1.03	1.04
x3	601.38	0.82	0.82	0.82
i8	940.68	0.69	0.62	0.6
x1	567.9	0.67	0.67	0.67
pair	1243.8	0.75	0.70	0.70
rot	1362.06	0.82	0.80	0.80
C6288	5971.5	0.94	0.86	-
des	1092.24	0.89	0.79	-
too_large	1069.2	0.52	0.54	0.54
AVERAGE		0.756	0.710	-

TABLE II
DELAY COMPARISON OF GENERALIZED AND TRADITIONAL BUFFERED PTL

The area calculation was performed by determining the active area of the MUXes and all the library cells. Table III compares the active area of the traditionally buffered partitioned ROBDD based implementation with that of the generalized buffering methodology (with and without CODCs). In this table, Column 1 lists the example under consideration. Column 2 and 3 reports the circuit active area for the traditional and generalized buffering without CODCs. Columns 4 and 5 report the area for the new PTL synthesis algorithm with generalized buffering using ACODCs and CODCs respectively. The area in Columns 3, 4 and 5 are expressed as a fraction of the area of the traditional method. Since the generalized buffers occupy a greater area than the traditional buffers, the overall area improvement of our method is not as high (5% on average) compared to traditional method. Our method also yields a 2% improvement in area (over generalized buffering without CODCs). In the case of the circuit C432, when CODCs were used, the area increases by 17% compared to the area when ACODCs were used. This increase in area is due to the use of OR4 and OR3 gates during Boolean division (when CODCs were used). When ACODCs were used, these gates were not utilized. Due to this, the CODC based method exhibits a 1% area overhead over ACODC based method (calculated over the examples for which the CODC method completed).

	Traditional	Generalized Buffering		
	Buffering	Without CODC's	With ACODCs	With CODCs
Ckt	Area (μ^2)	Area	Area	Area
alu2	164.32	0.87	0.86	0.83
alu4	963.68	1.12	1.12	-
apex6	305.52	0.95	0.93	0.93
C432	87.12	1.02	0.93	1.10
C499	106.24	0.92	0.92	0.92
C880	136.16	0.80	0.79	0.79
C1908	152.24	0.97	0.93	0.93
C3540	532.88	1.01	0.98	-
C5315	540.16	1.12	1.11	1.12
x3	315.76	0.96	0.96	0.96
i8	520.48	0.75	0.72	0.70
x1	120.88	0.96	0.95	0.95
pair	640.32	1.08	1.07	1.06
rot	229.20	0.96	0.96	0.96
C6288	1220.48	1.10	1.06	-
des	1808.88	0.94	0.91	-
too_large	125.68	0.98	0.97	0.97
AVERAGE		0.970	0.950	-

TABLE III
AREA COMPARISON OF GENERALIZED AND TRADITIONAL BUFFERED PTL

The number of MUXes is simply the sum of the sizes of each of the partitioned ROBDDs in the design. Tables IV reports the number of MUXes used for PTL synthesis using traditional and generalized buffering. Column 1 lists the example under consideration. Columns 2 reports the number of MUXes required for the traditional buffering. Columns 3 reports the number of MUXes used by generalized buffering without CODCs (as a fraction of the corresponding numbers for the traditional method). Columns 4 and 5 report the number of MUXes used for generalized buffering using ACODCs and CODCs respectively (again as a fraction of the corresponding numbers for the traditional method). We observe that our method utilizes about 27% fewer MUXes than traditional buffering and 4% fewer MUXes than generalized buffering without CODCs.

Table V reports the run-time for the different PTL synthesis algorithms with generalized buffering. In this table, Column 1 lists the circuit under consideration while Column 2 reports the run-time for generalized buffering without CODCs. Columns 3 and 4 report the run-time for generalized buffering using ACODCs and CODCs respectively. We observe from the Table V that the run-time taken generalized buffering with ACODCs is 8x more than the run-time for generalized buffering without CODCs. The run-time of our partitioning algorithm with ACODCs is slightly less than 152 seconds for the largest example in our benchmark suite. The run-time for complete CODCs was on average 76X that the run-time for ACODCs.

These results indicate that generalized buffering with

	Traditional	Generalized Buffering		
	Buffering	Without CODC's	ACODCs	CODCs
Ckt	MUX	MUX	MUX	MUX
alu2	718	0.600	0.577	0.532
alu4	4188	0.704	0.689	-
apex6	1337	0.737	0.719	0.718
C432	404	0.995	0.795	0.834
C499	404	0.762	0.762	0.762
C880	594	0.731	0.702	0.712
C1908	660	0.858	0.806	0.792
C3540	2362	0.732	0.672	-
C5315	2366	1.068	1.037	1.03
x3	1393	0.797	0.789	0.791
i8	2418	0.483	0.449	0.439
x1	527	0.666	0.657	0.652
pair	2925	0.790	0.726	0.718
rot	999	0.803	0.796	0.79
C6288	5393	0.917	0.834	-
des	7854	0.796	0.740	-
too_large	552	0.654	0.643	0.644
AVERAGE		0.770	0.729	-

TABLE IV
NUMBER OF MUXES UTILIZED DURING TRADITIONAL AND GENERALIZED BUFFERING

	Generalized Buffering		
	Without CODC's	With ACODCs	With CODCs
Ckt	Time(s)	Time(s)	Time(s)
alu2	0.380	11.48	435.32
alu4	8.990	54.52	-
apex6	1.490	4.6	117.23
C432	0.460	13.82	236.820
C499	0.360	11.67	86.78
C880	0.240	2.41	70.0
C1908	0.790	8.08	362.12
C3540	5.010	41.56	-
C5315	30.840	29.2	9543.8
x3	1.400	4.12	104.27
i8	1.630	28.73	2651.48
x1	0.240	1.08	16.2
pair	3.720	25.85	7110.52
rot	0.850	4.59	332.39
C6288	10.790	151.49	-
des	14.730	140.71	-
too_large	0.310	2.89	78.33
AVERAGE	4.837	33.55	-

TABLE V
RUN-TIME FOR PTL SYNTHESIS WITH GENERALIZED BUFFERING

ACODCs offers significantly faster designs, with a small area benefit as well, compared to traditionally and generalized (without CODCs) buffered PTL. *The use of complete CODCs provides minimal benefits over ACODCs. At the same time, the complete CODC computation took longer than the ACODCs on average*. Also, complete CODCs cannot be computed for larger circuits whereas ACODCs can be computed for arbitrary sized circuits. Therefore, the use of complete CODCs does not provide any practical benefits over ACODCs.

The effectiveness of our method can be augmented by invoking dynamic variable re-ordering while performing ROBDD construction. This will allow further reduction in circuit size for both the traditional and generalized buffering methodologies.

V. CONCLUSIONS

In this paper, we have presented a Pass Transistor Logic (PTL) circuit synthesis scheme. In order to handle larger designs, and also to ensure that the total number of series devices in the resulting circuit is bounded, partitioned Reduced Ordered Binary Decision Diagrams (ROBDDs) have been used to generate the PTL circuit. Our approach utilizes partitioned ROBDDs to construct the PTL circuit, with the interfaces between these PTL structures buffered using library gates. Our technique of *generalized* buffering uses Boolean division of the PTL block using different gates in a library. In this way, we can select the gate that results in the largest reduction in the height of the PTL block. In this manner, we can have these gates serve the function of buffering the outputs of PTL blocks, and also perform circuit computations at the same time. Approximate Compatible observability don't cares (ACODCs) or complete compatible observability don't cares (CODCs) were used along with Boolean division to simplify the PTL structures. Over a number of examples, we demonstrate that on average, our approach (generalized buffering with ACODCs) results in a 29% reduction in delay, and a 5% improvement in circuit area, compared to a traditional buffered PTL implementation. Our approach also resulted in reduction in delay by 5% and reduction in area by 2% over generalized buffering without don't cares.

The use of complete CODCs provides minimal benefits over ACODCs. At the same time, the complete CODC computation took significantly longer than the ACODCs. Additionally, complete CODCs can only be computed for circuits with up to a few thousand gates, while ACODCs can be computed for arbitrary sized circuits. Therefore, it is better to use ACODCs over complete CODCs. Results show that ACODCs provide enough don't cares to yield significantly better PTL structures in terms of area and delay.

REFERENCES

[1] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proc. of the Design Automation Conf.*, pp. 40–45, June 1990.

[3] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, *et al.*, "VIS: A system for verification and synthesis," in *International Computer-Aided Verification Conference* (R. Alur and T. A. Henzinger, eds.), vol. 1102, (New Brunswich, NJ), pp. 428–432, Springer-Verlag, July–August 1996.

[4] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[5] N. Saluja and S. Khatri, "A robust algorithm for approximate compatible observability don't care (codc) computation," in *Proceedings of the Design Automation Conference*, June 2004.

[6] P. Buch, A. Narayan, A. Newton, and A. Sangiovanni-Vincetelli, "Logic synthesis for large pass transistor circuits," in *Proceedings, IEEE/ACM International Conference on Computer-Aided Design*, pp. 663–670, Nov 1997.

[7] Hidden for blind review.

[8] K. Taki, “A survey for pass-transistor logic technologies-recent researches and developments and future prospects,” in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 223–226, Feb 1998.

[9] K. Yano, Y. Sasaki, K. Rikino, and K. Seki, “Top-down pass-transistor logic design,” *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 792–803, June 1996.

[10] L. Macchiarulo, L. Benini, and E. Macii, “On-the-fly layout generation for ptl macrocells,” in *Proceedings, Design, Automation and Test in Europe Conference*, pp. 546–551, March 2001.

[11] D. Radhakrishnan, S. Whitaker, and G. Maki, “Formal design procedures for pass transistor switching circuits,” *IEEE Journal of Solid-State Circuits*, vol. 20, pp. 531–536, April 1985.

[12] E. McCluskey, *Logic design principles : with emphasis on testable semicustom circuits*. Prentice-Hall, 1986.

[13] W. Quine, “The problem of simplifying truth functions,” *American Math. Monthly*, vol. 59, pp. 521–531, 1952.

[14] E. McCluskey, “Minimization of Boolean functions,” vol. 35, pp. 1417–1444, 1956.

[15] T. Cheung, K. Asada, and H. Wong, “Design automation algorithms for regenerative pass-transistor logic,” in *Proceedings of 1997 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, pp. 1540–1543, June 1997.

[16] S.-F. Hsiao, J.-S. Yeh, and D.-Y. Chen, “High-performance multiplexer-based logic synthesis using pass-transistor logic,” in *Proceedings of 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2, (Geneva), pp. 325–328, May 2000.

[17] J. Neves and A. Albicki, “A pass transistor regular structure for implementing multi-level combinational circuits,” in *Proceedings, Seventh Annual IEEE International ASIC Conference and Exhibit*, pp. 88–91, Sept 1994.

[18] C. Pedron and A. Stauffer, “Analysis and synthesis of combinational pass transistor circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 775–786, July 1988.

[19] D. Markovic, B. Nikolic, and V. Oklobdzija, “General method in synthesis of pass-transistor circuits,” in *Proceedings. 22nd International Conference on Microelectronics*, vol. 2, pp. 695–698, May 2000.

[20] S. Shelar and S. Sapatnekar, “An efficient algorithm for low power pass transistor logic synthesis,” in *Proceedings, Asia and South Pacific Design Automation Conference and the 15th International Conference on VLSI Design*, pp. 87–92, Jan 2002.

[21] Y.-T. Lai, Y.-C. Jiang, and H.-M. Chu, “BDD decomposition for mixed CMOS/PTL logic circuit synthesis,” in *Proceedings, IEEE International Symposium on Circuits and Systems*, pp. 5649–5652, May 2005.

[22] S. Yamashita, K. Yano, Y. Sasaki, Y. Akita, H. Chikata, K. Rikino, and K. Seki, “Pass-transistor/CMOS collaborated logic: The best of both worlds,” in *Digest of Technical Papers, Symposium on VLSI Circuits*, pp. 31–32, June 1997.

[23] R. Garg and S. Khatri, “Generalized buffering of ptl logic stages using boolean division,” in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2006. To appear.

[24] L. Nagel, “Spice: A computer program to simulate computer circuits,” in *University of California, Berkeley UCB/ERL Memo M520*, May 1995.

[25] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu, “New paradigm of predictive MOSFET and interconnect modeling for early circuit design,” in *Proc. of IEEE Custom Integrated Circuit Conference*, pp. 201–204, Jun 2000. http://www-device.eecs.berkeley.edu/ ptm.