BDD-based Two Variable Sharing Extraction

Dennis Wu, Jianwen Zhu

Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
{wudenni, jzhu}@eecg.toronto.edu

Abstract

It has been shown that Binary Decision Diagram (BDD) based logic synthesis enjoys faster runtime than the classic logic synthesis systems based on Sum of Product (SOP) form. However, its synthesis quality has not been on par with the classic method due to the lack of an effective sharing extraction strategy. In this paper, we present the first sharing extraction algorithm that directly exploits the structural properties of BDD. While our sharing extraction algorithm is limited to two-variable, disjunctive factors, and therefore may miss sharing opportunities, we show that it can be made exact, incremental and polynomial. Our experimental results under a comprehensive BDD-based synthesis tool show that this technique inflates runtime by a mere 6% while enabling area savings of over 25%.

1. Introduction

Logic synthesis, the task of optimizing gate level networks, has been the corner stone of modern electronic design automation methodology since the 1990s. As the chip size grows exponentially, and as the logic synthesis task increasingly becomes coupled with physical design, the synthesis runtime has emerged as a new priority, in addition to the traditional metrics of synthesis quality, including area, speed and power. To this end, there is a growing interest in migrating from an algebraic method, exemplified by SIS [7], to a Binary Decision Diagram (BDD) based method, exemplified by BDS [8]. Compared with the former, which uses cube set as the central data structure for design optimization, the latter exploits the compactness and canonicality of BDD so that Boolean decomposition, Boolean matching and don't care minimization can be performed in an efficient way. Despite these advantages, our experiments on publicly available packages show that BDD-based methods are not yet competitive with cube set based methods in terms of area quality.

A major reason for this shortcoming is the lack of a sharing extraction strategy. Sharing extraction is the process of extracting common functions among gates in the Boolean network to save area. Their usefulness have long been proven in cube set based systems. One example implementation is kernel extraction, which has been central in producing low area designs in the SIS [7] synthesis package and commercial tools. In contrast, BDD-based systems have provided relatively low support for sharing extraction.

In this paper, we show the first sharing extraction algorithm that directly exploits the structural properties of BDDs. More specifically, we make the following contributions. First, we demonstrate that by limiting our attention to a specific class of extractors (similar to limiting to kernels in the classic method), namely two-variable disjunctive extractors, effective area reduction can be achieved. Second, we show that an exact, polynomial time algorithm can be developed for the full enumeration of such extractors. Third, we show that just like the case of kernels, there are inherent structures for the set of extractors contained in a logic function, which we can use to make the algorithm incremental and as such, further speed up the algorithm. Our experiments indicates that an overhead of merely 6% is needed to run our sharing extraction algorithm, whereas 25% area reduction can be achieved. As the result, our logic synthesis system performs consistently better than state-of-the-art BDD synthesis packages both in terms of runtime and synthesis quality.

The remainder of the paper is divided into eight sections. Section 2 introduces related works. Section 3 gives an overview of the extraction flow. Section 4 describes the extraction problem for arbitrary functions. Section 5 describes the extraction problem for two-variable, disjunctive extractors. Section 6 gives an incremental solution to finding good extractors. Section 7 uses a heuristic based on the transitive property of good extractors to further improve runtime. In Section 8, experimental results are presented before we conclude the paper in Section 9.

2. Related Works

Perhaps the most widely used sharing extraction algorithm is the cubeset based kernel extraction by Brayton and McMullen [1]. Their algorithm works by enumerating candidate factors, for all gates, followed by selecting the factor that generates the most area reduction, as measured by their size and number of repetitions. Their factorizations take the form $F = AB + C$, where $supp(A)$ and $supp(B)$ are disjoint. Here they make the simplification, that a variable and it's compliment are treated as two independent variables, in order to make the algorithm fast. Their sharing extraction algorithm is *active*, because at each step, an attempt is made to extract the best sharing opportunity. In contrast, *passive* sharing extraction finds sharing only after a complete decomposition is performed.

Sawada et al [6] describe a BDD-based equivalent for kernel extraction. While they use BDDs to represent logic functions, they are represented in Zero-Suppressed Decision Diagram (ZDD) form, which implicitly represents cubesets. Essentially, the algorithm is cubeset based and cannot use the advantages of the BDD as described earlier.

A subproblem of sharing extraction, and one that has garnered the most attention in BDD-based systems, is decomposition. The purpose of decomposition, like sharing extraction, is to break large gates down into smaller ones. It differs in that decompositions are judged by area savings with respect to a single gate, without considering external opportunities for sharing. Because of it's strength in decomposition, BDD-based synthesis systems often perform decomposition first and then apply a passive form of sharing extraction.

BDS [8] takes an approach to synthesis that moves away from cubesets altogether. They identify good decompositions by relying heavily on structural properties of BDDs. For example, 1, 0 and X dominators produce algebraic AND, OR and XOR decompositions respectively. They also describe structural methods for non-disjunctive decomposition based on their concept of a generalized dominator. They also perform other non-disjunctive decompositions, such as variable and functional mux decompositions. After performing a complete decomposition of the circuit, they perform sharing extraction by computing BDDs for each node in the Boolean network, in terms of the primary inputs. Nodes with equivalent BDDs can be shared. For obvious reasons, this passive form of sharing extraction produces sharing results inferior to kernel extraction.

Mishchenko et al [3] developed a BDD-based synthesis system centered on the Bi-decomposition of functions. They give a theory for when strong or weak bi-decompositions exist and give expressions for deriving their decomposition results. Their sharing extraction step is interleaved with decomposition so that sharing can be found earlier, avoiding redundant computations. They also retain don't care information across network transformations to increase flexibility in matching. However, their's is still a passive sharing extraction.

3. Overview

Our sharing extraction algorithm, like kernel extraction, decomposes sharing extraction into a two-step flow. In the first step, the candidate extractors are *enumerated* for each gate in the network. For practicality, not *all* extractors can be enumerated because they are too numerous. In kernel extraction, extractors are limited to those of the algebraic kind, because they can be found efficiently on the cube set. Similarly, we limit our extractors to two variable, disjunctive extractors because they can be found efficiently on the BDD.

In the second step, common extractors are *selected* to share. Committing some extractors destroy others so ordering is important in choosing the extractors that have the most impact. One method that works well, is to select the extractors greedily, based on the size of the extractor and the number of times the extractor is repeated (frequency). In two variable extraction, all extractors have size of two, so selection is based solely on the frequency of the extractor. Extractors are matched in a hash table using a key based on their two variable support and gate type.

With the selection step described, the remainder of the paper will focuses on the process of enumerating extractors. We use the following conventions for notations. Uppercase letters F, G, H represent functions. Lowercase letters a, b, c represent the variables of those functions. $Supp(F)$ is the support set of F. $F|_x$ is the cofactor of F with respect to x. $[F, C]$ represents an incompletely specified function with F as it's completely specified function and C as it's care set. $\Downarrow$ represents the restrict operation.

4. Functional Extraction

Given two functions F and E, the extraction process breaks F into two simpler functions, extractor E and remainder R.

$$\begin{aligned} F(X) &= R(e, X_R) && (1)\\ R(e, X_R) &= eR_1(X_R) + \overline{e}R_2(X_R) && (2)\\ e &= E(X_E) && (3)\end{aligned}$$

X is the support set of F. X_E is the support set of E. X_R is the support set of R. $X_E \bigcup X_R = X$.

Both R_1 and R_2 have multiple solutions. The range of solutions can be characterized by an incompletely specified function $[F, C]$, where F is a completely specified solution and C is the care set. One solution is $R_1 = F$ and $R_2 = F$. We obtain the C conditions by noting R_1 is a don't care when E is false and R_2 is a don't care when E is true.

$$\begin{aligned} R_1 &= [F, E] && (4)\\ R_2 &= [F, \overline{E}] && (5)\end{aligned}$$

We want a completely specified solution that minimizes the complexity of R_1 and R_2. To do this, we assign the don't care conditions in a way that minimizes the resulting node count. This problem was found to be NP complete [5] but a solution can be obtained using one of several don't care minimization heuristics. One well known heuristic, which has been shown to be fast, is the restrict operation [2]. Applying the restrict operator, the final equations for the remainder and extractor are shown below:

$$\begin{aligned} R(e, X_R) &= e(F \Downarrow E) + \overline{e}(F \Downarrow \overline{E})\\ e &= E(X_E)\end{aligned}$$

5. Disjunctive Two Variable Extraction

The last section described how to compute the remainder for an arbitrary function and extractor. In this section we describe a specialized extraction algorithm tailored to good extractors.

DEFINITION 1. *Given function F, extractor E and remainder R, good extractors are two variable extractors whose variables are disjunctive from R. E and R are disjunctive when they do not share support.*

It is important to note, the limitations of good extractors will force us to miss some good sharing opportunities. Not all good sharing opportunities use disjunctive extractors. Nor are all disjunctive extractors the combination of two variable disjunctive extractors. Restricting candidate extractors is necessary however, to keep the runtime reasonable. Nevertheless, good extractors are good candidates because they can be found and matched quickly. We show experimentally that they are effective in reducing area.

5.1 Extractor Types

All two variable functions are considered potential good extractors. A two variable function has four unique input values. Each of these input values have two possible outputs. That makes $4^2 = 16$ unique, two variable, functions. The one and zero constants and the single variable functions ($F = a$, $F = \overline{a}$, $F = \overline{b}$ and $F = \overline{b}$) make six trivial functions. These functions cannot produce good extractions. The ten remaining functions are listed below:

Condition	Extractor	Remainder
$F\|_{a\overline{b}} = F\|_{\overline{a}b} = F\|_{\overline{a}\overline{b}}$	AND	$R = eF\|_{ab} + \overline{e}F\|_{a\overline{b}}$
$F\|_{ab} = F\|_{a\overline{b}} = F\|_{\overline{a}b}$	OR	$R = eF\|_{ab} + \overline{e}F\|_{\overline{a}\overline{b}}$
$F\|_{ab} = F\|_{\overline{a}b} = F\|_{\overline{a}\overline{b}}$	AND10	$R = eF\|_{a\overline{b}} + \overline{e}F\|_{ab}$
$F\|_{ab} = F\|_{a\overline{b}} = F\|_{\overline{a}\overline{b}}$	AND01	$R = eF\|_{\overline{a}b} + \overline{e}F\|_{ab}$
$F\|_{ab} = F\|_{\overline{a}\overline{b}}$ & $F\|_{a\overline{b}} = F\|_{\overline{a}b}$	XOR	$R = eF\|_{\overline{a}b} + \overline{e}F\|_{ab}$

Table 1: Cofactor conditions for good extraction

ALGORITHM 1. *Finding Good Extractors*

```
findExtractors( F ) {
  forall( pairs of variables (a,b) )  {
    $A = F|_{ab}$;
    $B = F_{\overline{a}b}$;
    $C = F|_{a\overline{b}}$;
    $D = F|_{\overline{a}\overline{b}}$;

    if( $B = C = D$ )
      // Found good AND ($ab$) extractor.
    else if( $A = B = C$ )
      // Found good OR ($a$+b)extractor.
    else if( $A = B = D$ )
      // Found good $a\overline{b}$ extractor.
    else if( $A = C = D$ )
      // Found good $\overline{a}b$ extractor.
    else if( $A = D$ and $B = C$ )
      // Found good XOR ($a \oplus b$) extractor.
 }}
```

$$
\begin{aligned}
F &= ab & F &= \overline{a} + \overline{b} \\
F &= a + b & F &= \overline{a}\overline{b} \\
F &= a \oplus b & F &= a \overline{\oplus} b \\
F &= a\overline{b} & F &= \overline{a} + b \\
F &= \overline{a}b & F &= a + \overline{b}
\end{aligned}
$$

The right five functions are compliments of the left five. They will produce the same extractions so half can be discarded. In total, there are five functions to consider when looking for good extractions.

5.2 Computing Extraction

The same procedure used for computing extraction for arbitrary functions and extractors (shown earlier) can also be applied to good extractors. However, a faster algorithm is available for the special case of good extractors. Essentially, good extractors require equivalence between certain cofactors of F.

THEOREM 1. *$E = ab$ is a good extractor of F iff $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.*

The theorem states that there is a disjunctive, two variable, AND extractions can be detected by comparing three cofactors for equivalence. Cofactor conditions also exist for the four other extractor types, and are listed in Table 1.

The complete extraction search algorithm is shown in Algorithm 1. The for loop iterates $O(N^2)$ times, where N is the number of variables, and each time performs a $O(G)$ cofactor operation. Thus the total worst case complexity for finding the good extractors of a function is $O(N^2G)$.

6. Incrementally Finding Extractors

In this section we discuss techniques that speed up the extraction algorithm further. The first improvement uses the property that good extractors of a function continue to be good extractors in their remainders. Instead of rediscovering these good extractors, they can be copied over.

THEOREM 2. *Let E_1 and E_2 be arbitrary good extractors of F. $Supp(E_1) = \{a, b\}$, $Supp(E_2) = \{c, d\}$ and $Supp(E_1) \bigcap Supp(E_2) = \oslash$. If R is the remainder of F extracted by E_1, then E_2 is a good extractor of R.*

We call these extractors "copy" extractors. Copy extractors do not account for *all* good extractors of R. The good extractors missed are those formed with variable e. To find these extractors, cofactor conditions between e and every other variables of R must be checked. Extractors found in this way are called "new e" extractors. These two types of extractors, in fact, account for all good extractors of R. The benefit is that good extractors of R can be obtained through "copy" and "new e" extractors. This is faster than computing good extractors directly.

THEOREM 3. *Let R be the remainder of F extracted by E_1. E is a good extractor of R iff E is a "copy" extractor or "new e" extractor.*

The complexity of transferring extractors from F to R is $O(N^2)$. The complexity for finding new extractors involving variable e is $O(NG)$. The total complexity for finding extractors for a remainder is $O(N^2 + NG)$. The incremental algorithm only applies when finding extractors for *remainders*. When finding extractors for functions whose parent extractors have not been computed, the $O(N^2G)$ complexity still applies.

7. Transitive Property of Good Extractors

The $O(N^2G)$ complexity required to find the initial set of extractors can be reduced if we are willing to relax the condition that all good extractors be found.

THEOREM 4. *$E_1(a, b)$ and $E_2(b, c)$ are good extractors of $F \Rightarrow \exists\, E_3(a, c)$ such that $E_3(a, c)$ is a good extractor of F.*

The transitive property of good extractors allows us to reduce the complexity of finding good extractors. In our previous algorithm, the $O(N^2G)$ complexity arose from the need to explicitly find extractors between every pair of variables. Using the transitive property of extractors, we only look for extractors between variables that are adjacent in the BDD order. This reduces the number of pairs we consider from $O(N^2)$ to $O(N)$. The transitive property then, is applied across successively adjacent extractors to find additional extractors. The new algorithm relies on a heuristic: If two variables a and b form a good extractor, then they are likely to satisfy one of two conditions:

1. They are adjacent in the BDD variable order.
2. They are separated by variables that form good extractors with their adjacent variables.

This is not a rule however, as good extractors can be formed that do not satisfy the above conditions. The heuristic works well however, because variables that form good extractors are likely clustered together in the BDD variable order; It reduces node count. What we have is a trade off between finding all extractors, and finding them quickly. In our experimental results however, the tradeoff in using this heuristic is minimal, degrading area quality by only 0.1%.

Table 2: Area results for MCNC benchmark.

Circuit	XY		XY (NSE)		SIS		BDS	
C1355	15820		16537.5	4.5%	15855.0	0.2%	16222.5	2.5%
C1908	15802.5		19897.5	25.9%	15715.0	-0.6%	16502	4.4%
C2670	20965		22382.5	6.8%	22225.0	6.0%	22645	8.0%
C3540	37292.5		43382.5	16.3%	36977.5	-0.9%	39165	5.0%
C5315	54705		61530.0	12.5%	52395.0	-4.4%	61565	12.5%
C6288	95602.5		95252.5	-0.4%	90265.0	-5.9%	109130	14.1%
C7552	67462.5		67427.5	-0.1%	67060.0	-0.6%	70805	5.0%
alu4	26162.5		27020.0	3.3%	21892.5	-19.5%	29960	14.5%
dalu	33827.5		39637.5	17.2%	28997.5	-16.7%	42122.5	24.5%
des	137970		148715.0	7.8%	110582.5	-24.8%	176680	28.1%
frg2	29732.5		40512.5	36.3%	26390.0	-12.7%	36435	22.5%
i10	76580		88760.0	15.9%	71242.0	-7.5%	81235	6.1%
i8	32567.5		48212.5	48.0%	30975.0	-5.1%	39760	22.1%
i9	18060		25322.5	40.2%	18060.0	0.0%	20317.5	12.5%
k2	34510		81550.0	136.3%	34160.0	-1.0%	failed!	
pair	57750		58310.0	1.0%	52395.0	-10.2%	57470	-0.5%
rot	22435		23922.5	6.6%	22120.0	-1.4%	25095	11.9%
t481	1277.5		1277.5	0.0%	12057.5	843.8%	1277.5	0.0%
too_large	26337.5		31727.5	20.5%	8855.0	-197.4%	55492.5	110.7%
vda	18567.5		42280.0	127.7%	18987.5	2.3%	25060	35.0%
x3	29697.5		31552.5	6.2%	25322.5	-17.3%	29872.5	0.6%
AVG				25.4%		25.1%		17.0%

Table 3: Run time results for MCNC benchmark.

Circuit	XY		XY(NSE)		SIS		BDS	
C1355	4010		3290	-21.9%	4200	4.7%	2850	-40.7%
C1908	7210		6180	-16.7%	5100	-41.4%	5310	-35.8%
C2670	4290		3860	-11.1%	117900	2648.3%	2470	-73.7%
C3540	15400		12360	-24.6%	52200	239.0%	6920	-122.5%
C5315	6060		5910	-2.5%	7800	28.7%	8990	48.3%
C6288	16530		16460	-0.4%	18200	10.1%	10400	-58.9%
C7552	57170		26420	-116.4%	32400	-76.5%	25250	-126.4%
alu4	8900		9920	11.5%	53900	505.6%	8870	-0.3%
dalu	6420		4030	-59.3%	36200	463.9%	8520	32.7%
des	26210		65160	148.6%	58500	123.2%	35830	36.7%
frg2	5980		4600	-30.0%	9400	57.2%	9930	66.1%
i10	21110		39020	84.8%	172000	714.8%	17640	-19.7%
i8	11740		12100	3.1%	9400	-24.9%	14220	21.1%
i9	4360		2930	-48.8%	3000	-45.3%	6640	52.3%
k2	112500		249790	122.0%	23500	-378.7%	failed!	
pair	8280		7370	-12.3%	9300	12.3%	18220	120.0%
rot	4470		5010	12.1%	3600	-24.2%	15540	247.7%
t481	9140		6160	-48.4%	33400	265.4%	7750	-17.9%
too_large	31040		38280	23.3%	1976000	6266.0%	190840	514.8%
vda	10700		23160	116.4%	11100	3.7%	9270	-15.4%
x3	3200		3030	-5.6%	2800	-14.3%	5530	72.8%
AVG				5.9%		511.3%		35.1%

8. Experimental Results

To evaluate our proposal we implemented a complete logic synthesis system that includes the sharing extraction algorithm described in this paper. We perform many of the steps present in typical synthesis flows. *Logic minimization* is performed through BDD variable reordering using the sifting heuristic [4]. The sifting heuristic, like bubble sort, swaps adjacent variables in search of the minimum BDD size. The *sweep* stage, further simplifies gates by propagating constant values, and merging support that is repeated more than once within a single gate. In the *elimination* stage, gates of the network are selectively collapsed in an attempt to remove inter-gate redundancies. Finally, the *sharing extraction* and *decomposition* steps are interleaved. Sharing extraction is applied first to find as much disjunctive sharing as possible. Decomposition then breaks down gates where sharing extraction cannot, such as where conjunctive decompositions are required. As decompositions are applied, new good extractors may be created and sharing extraction is re-applied.

The experiments were conducted on a dual processor Solaris Blade 1000 with 2.5 GB memory running SunOS version 5.8. The benchmarks were taken from the combinational multi-level examples of MCNC91. Only those circuits that were reported by [9] to have an approximate gate count of 500 or more were selected for testing. We put the benchmarks through three synthesis systems for comparison. We run our synthesis system with an without sharing extraction. These are labeled XY and XY(NSE) respectively in Table 2. We also run the benchmark on BDS, a BDD-based synthesis system, and SIS a cube set based synthesis system. The optimized results are then mapped to a 0.35um CMOS TSMC standard cell library using Synopsis Design Compiler.

Table 2 shows the area produced for each synthesis system in squared micrometers. Percent differences compared to XY (with sharing extraction is calculated as

$$pd = (area - XY\_area)/min(area, XY\_area).$$

The area saving from sharing extraction is quite substantial, reducing area by 25% over XY without sharing extraction. Compared with BDS, XY produces circuits with 17% less area. XY, on average, produces circuits with 25% less area than SIS. However, this result is heavily skewed by one circuit, t481. In fact, SIS produced better results for the majority of circuits.

In terms of runtime, XY runs over 500% faster than SIS. It runs 6% slower than the version of XY without sharing extraction, and runs 35% faster than BDS. The run time results are shown in Table 3.

9. Conclusions

In this paper, two variable disjunctive extractors are used as candidates for sharing extraction. With the BDD, these extractors can be found quickly by computing the result of a few, single cube, cofactors. Old, good extractors remain valid in their remainder functions, enabling a fast, incremental solution. And good extractors are transitive, enabling us to reduce the search to extractors of adjacent variables, while sacrificing very little in terms of area quality.

References

[1] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *ISCAS Proceedings*, pages 49–54, 1982.

[2] 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*, pages 365–373, 1989.

[3] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proceeding of the 38th Design Automation Conference*, pages 103–108, 2001.

[4] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

[5] M. Sauerhoff and I. Wegener. On the complexity of minimizing the OBDD size of incompletely specified functions. In *IEEE Transactions on Computer Aided Design*, pages 1434–1437, 1996.

[6] H. Sawada, S. Yamashita, and A. Nagoya. An efficient method for generating kernels on implicit cube set representations. In *International Workshop on Logic Synthesis*, 1999.

[7] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanaha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuits synthesis. Technical Report UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, 1992.

[8] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proceeding of the 37th Design Automation Conference*, pages 92–97, 2000.

[9] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. Technical report, Microelectronics Center of North Carolina, P. O. Box 12889, Research Triangle Park, NC 27709, 1991.