Towards Scalable BDD-Based Logic Synthesis

by

Dennis Wu

A thesis submitted in conformity with the requirements
for the degree of Masters of Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Towards Scalable BDD-Based Logic Synthesis

Dennis Wu

Masters of Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

The past decade of logic synthesis research has looked at using Binary Decision Diagrams (BDDs) as an alternative to the traditional sum-of-product representation of logic functions. When compared to the later, logic synthesis algorithms using BDDs have been shown to have significantly better scalability, however, the area quality produced has been poor. This thesis describes two new improvements to BDD-based logic synthesis. The first is a sharing extraction algorithm to improve area. The second is a logic folding approach, where equivalent logic transformations are shared to improve runtime.

The algorithms are evaluated in a new logic synthesis tool called FBDD. Experimental results on the MCNC benchmarks show an average area savings of 21% and runtime improvements of 3 times, when compared to a state-of-the-art BDD based logic synthesis system.

Acknowledgements

I would first like to thank my supervisor Jianwen Zhu for encouraging me to pursue my Master's, and introducing me to the field of computer aided design. My education has been enriched, in large part, by his mentoring.

I would like to thank Dr. Francis, Dr. Najm and Dr. Veneris for volunteering to be a part of my review committee, especially on such short notice.

I would also like to thank the members of my research group, Zhong, Fang, Linda, Rami and Silvian, for their valuable technical discussions during our group meetings. And I would like to thank Jinny and my family for their excessive encouragment.

Contents

List of Figures

List of Tables

Chapter 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 [19], to a Binary Decision Diagram (BDD) based method, exemplified by BDS [22]. 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 the difference in quality between BDD based and cubeset based synthesis 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 [19] synthesis package and commercial tools. In contrast, BDD based systems have provided relatively low support for sharing extraction.

In this thesis, we describe 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. As a result, our logic synthesis system performs consistently better than state-of-the-art BDD synthesis packages both in terms of runtime and synthesis quality.

To further improve runtime and scalability, we propose a logic synthesis approach that exploits the regularities commonly found in circuits. Regularities, are the repetition of logic functions. They are clearly abundant in array-based circuits such as arithmetic units, but can also be found in random logic as well. Using the simple metric of regularity, (# of logic components) / (# of logic component types), we typically found the regularity of datapath circuits to be on the order of several hundreds. It is interesting to note that it is usually the datapath circuit that blows up circuit complexity.

The introduction of the BDD, with it's fast equivalence checking properties, has made the fast detection of regularity in circuits possible. Equivalence checking, can now be performed among functions, without concern of function size, in constant time. Our logic synthesis system, aggressively applies this new capability throughout the entire synthesis flow. With regularity information at hand, logic transformations applied to one circuit structure can be easily shared wherever the logic structure is repeated. This dramatically reduces the number of logic transformations required for synthesis and improves runtime.

The rest of the thesis divided in to five chapters. In chapter 2, we introduce the BDD and describe how it relates to logic synthesis. Chapter 3 gives an exact BDD based sharing extraction algorithm, followed by a faster two variable version. In chapter 4 the regularity aware framework is described, along with examples of how it is applied to logic synthesis tasks. In chapter five we give experimental results. And finally, we conclude in chapter six.

Chapter 2

Background

2.1 Boolean Network

A Boolean Network is a directed acyclic graph whose nodes represent gates, and whose input and output edges represent the fanin and fanout of the gate respectively. In this thesis, we only deal with gates represented by completely specified, single output Boolean functions. A *Boolean function* is a mapping between the values of its n input signals to the value of its output signal $\{0,1\}^n \rightarrow \{1,0\}$. It is completely specified if each binary input vector maps to either a constant one or zero, as opposed to incompletely specified functions where the input vectors may correspond to an undefined (or don't care) output value.

Efficient representation of the Boolean function is central to building scalable and efficient logic synthesis systems. The first representation taught in any elementary course on digital logic, is the *Truth Table*. In the truth table, each record (or row) maps an input vector with an output value. All input vectors (or minterms) are written out to define the function. While effective for illustrating ideas, such as logic minimization using Kaurnough Maps, this representation has only been of theoretical importance because of poor scalability. Its exponential growth with respect to the number of inputs, makes functions of even moderate size impractical. A more practical represention, for the purpose of logic synthesis, is the *cubeset* representation. Here a third, don't care value, can be used to mark variables in the input vector of a record (called cubes in cubeset form). A variable marked as don't care indicates that the record's output value does not depend on that variable. For example, the input vectors 1011 and 1001 both produce a 1 output and can be rewritten as $10-1$. In practice, almost all classes of logic functions found in digital circuits can be efficiently represented using these don't cares. Furthermore, exact and efficient logic minimization algorithms (Esspresso) and sharing extraction algorithms (kernel extraction), that use cubesets as their logic function representation, have made cubesets the work horse of logic synthesis for the past two decades.

Recently there has been interest in another function representation, called Reduced Ordered Binary Decision Diagram (ROBDD or just BDD for short), introduced to the CAD community by Bryant [4]. The BDD has several attractive properties. It is canonical: function blocks that implement the same Boolean function and share the same support set have the same BDD representation. As a result, equivalent nodes in the BDD can be identified and collapsed easily to make the BDD compact. The BDD has also been shown to have several fast decomposition algorithms including XOR and Boolean decompositions that previous to the BDD, were difficult to perform [22].

Examples of the various representations for the function $F = \overline{a}\overline{b}\overline{c} + \overline{a}bc + ac$ is illustrated in Figure 2.1.

2.2 Binary Decision Diagram

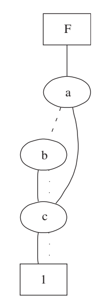
A Binary Decision Diagram is a directed, acyclic graph with a root node that represents a function, and terminal nodes '1' and '0' which are the outcomes of the function. The value of a function is determined by beginning at the root node and traversing down the graph, through a series of decisions, that lead to either the '1' or '0' terminal. Each node

a	b	c	s
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) TruthTable

a	b	c	s
0	0	0	1
0	1	1	1
1	-	1	1

(b) Cubeset



(c) Bianry Decision Diagram

Figure 2.1: Various Boolean Function Representations.

n in the graph contains three pieces of information; it is represented by one variable of the function $n.var$, and has two edges pointing to other nodes in the graph, $n.then$ and $n.else$. When evaluating the function at node n, the *then* edge ($n.then$) is traversed if the variable pointed to by $n.var$ is *true*, otherwise the *else* edge ($n.else$) is traversed. The BDD of $F = abc + a\overline{b}c + \overline{a}bc$ is shown in Figure 2.2. Dashed edges represent *else* edges, while solid lines represent *then* edges.

A Reduced Ordered Binary Decision Diagram (ROBDD) adds some restrictions to make the BDD compact and canonical. In the ROBDD, the variables have a total ordering that place all nodes of the same variable to a fixed level in the BDD. The ROBDD, as such, still grows exponentially with the number of inputs. Two reduction rules make the ROBDD compact. The first reduction rule removes redundant tests (Figure 2.3a). A node u with $u.then = u.else$ will traverse to $u.then$ regardless of the value of $u.var$. The node can be eliminated by redirecting nodes that point to u to

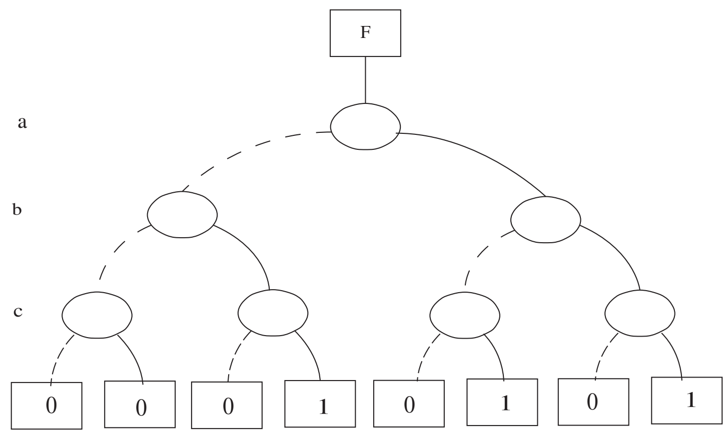
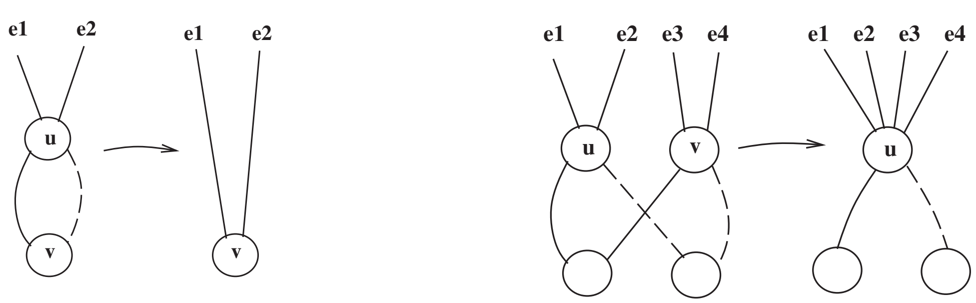


Figure 2.2: Binary Decision Diagram.

point to *u.then* (*u.else*) directly. The second reduction rule removes duplicate nodes (Figure 2.3b). Two nodes u, v that have the same *var*, *then* and *else* values produce the same function. v can be removed by redirecting edges pointing to v to u instead.



(a) Remove Redundant Tests.

(b) Remove Duplicate Nodes.

Figure 2.3: ROBDD Reduction Rules.

The ROBDD version of $F = abc + a\overline{b}c + \overline{a}bc$, shown in Figure 2.4, has a total of three nodes, down from the seven node version in Figure 2.2.

Modern day BDDs add a third type of edge called *compliment edges* which allow for further reductions in BDD size. Compliment edges differ from regular edges in that they return the compliment (instead of the regular) value of the BDD they point to. Compliment edges are restricted to being *else* edges. In total, BDDs are made up of

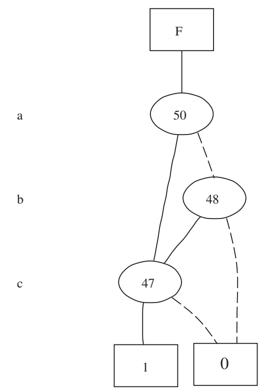


Figure 2.4: Reduced Ordered Binary Decision Diagram.

three edge types: *then* edges, regular *else* edges, and compliment *else* edges. The ROBDD version of $F = abc + a\overline{b}c + \overline{a}bc$ with compliment edges is shown in Figure 2.5. Compliment edges are represented by dotted lines. For brevity, ROBDDs with compliment edges are simply refered to as BDDs for the remainder of the thesis.

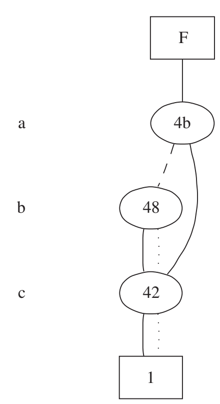


Figure 2.5: Reduced Ordered Binary Decision Diagram with Compliment Edges.

BDDs bring with them, several advantages over the traditional cubeset representation. First they are compact for more classes of functions than the cubeset. Both can represent algebraic gates such as AND and OR gates efficiently. However, cubesets have difficulty representing arithmetic intensive circuits such as XOR and MUX gates. The size of the cubeset grows exponentially with the number of inputs for XOR and MUX gates, while the size of the BDD grows linearly. Compact representations for arithmetic logic reduces memory consumption, and also translates into algorithms with faster runtimes.

A second advantage of BDDs is that under a common variable ordering, they are canonical. A representation is *canonical* if the representation for logic funtions is a unique. By constrast, there is more than one way to represent a function in cubeset form. Being canonical has the benefit of making equivalence checking extremely fast. Two functions, with the same variable order, will produce exactly the same BDDs and equivalence can be determined by a constant time pointer comparision. This new capability has opened opportunities in several areas including Boolean matching and verification, to name a few.

2.3 Logic Synthesis Flow

The Boolean network produced by a high level description language (HDL), such as Verilog or VHDL, is not suitable for direct circuit implementation. The Boolean network is subject to contain redundancies and gates with fan-ins that are too large. The purpose of logic synthesis is to transform this Boolean network into one that is composed of a realizable set of gates while optimizing some cost functions such as gate count and speed. Logic synthesis is a well studied problem with over 30 years of history. Our BDD based system follows a SIS-like synthesis flow which divides synthesis into four major tasks; Sweep, Eliminate, Logic Simplification, and finally Decomposition. Where BDD based systems differ, is in the way they perform each synthesis task. In this section we give a brief introduction to the logic synthesis process with specific attention given to BDD based algorithms.

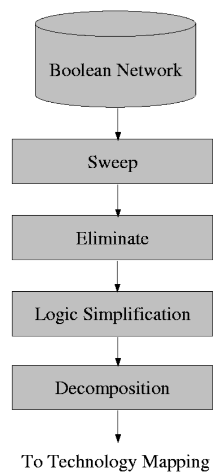


Figure 2.6: Basic synthesis flow.

2.3.1 Sweep

The *sweep* stage removes some obvious redundancies in the Boolean network by looking for two conditions. The first simplification, called constant node propagation, simplifies gates that are directly connected to power or ground. In Figure 2.7a, constant propagation simplifies a three input AND gate with a constant '1' at it's input into a two input AND gate. The second simplification, called common support merging, simplifies gates that have input variables repeated more than once in their support. In Figure 2.7b, common support merging simplifies a three input AND gate with a repeating input into a two input AND gate. These simplifications are easy to apply and are performed early in the synthesis flow to reduce the overall complexity of the circuit before more time consuming logic transformations are applied.

2.3.2 Eliminate

The *eliminate* stage attempts to remove inter-gate redundancies by merging adjacent, highly correlated gates together. Elimination is performed through trial and error by collapsing a gate into it's fannout. If the complexity after elimination is lower than the

(a) Constant Propagation. (b) Common Support Merging.

Figure 2.7: Sweep Operations.

complexity before elimination by some threshold, then the elimination is accepted. We use total BDD node count as the measure of complexity. In Figure 2.8, two two-input AND gates are transformed into a single two-input AND gate after elimination.

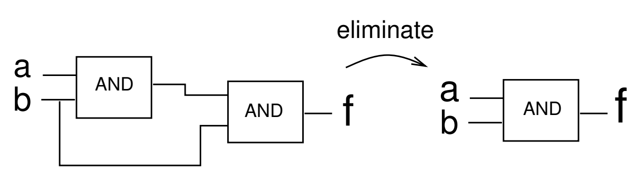


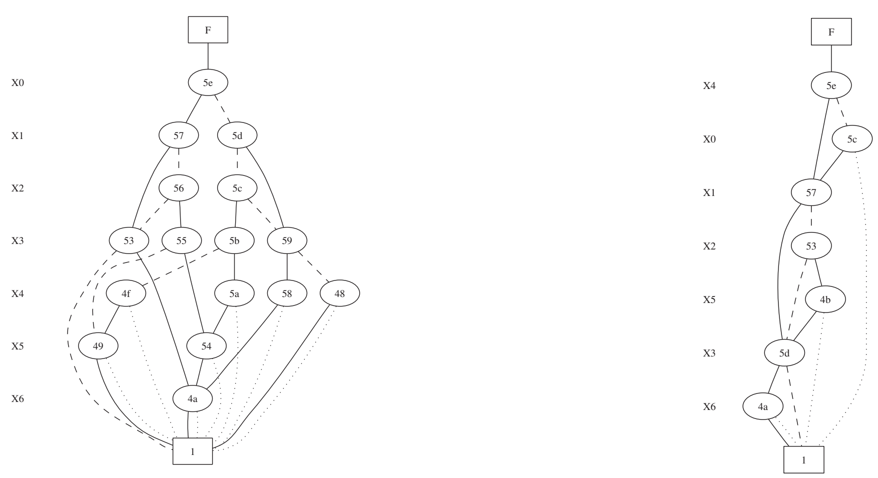
Figure 2.8: Elimination.

The size of most industrial circuits prevent function blocks from being completely collapsed into the primary outputs. Large BDDs can cause memory blow up: the number of nodes in completely collapsed networks are several orders of magnitude greater than their uncollapsed counterparts [22]. Also, the speed of variable ordering becomes prohibitive for large BDDs. A balance is needed between collapsing function blocks to remove inter-gate redundancies and keeping BDD sizes reasonable.

2.3.3 Simplification

Once the function blocks have been collapsed, the Boolean networks are *simplified* through variable reordering. This is similar to the two-level minimization of cubesets performed in SIS. Variable reordering reduces the number of nodes in a BDD through iterative, variable swapping techniques. Because variable ordering is slow and may need to be applied several times during logic synthesis, variable ordering can contribute to a significant portion of the runtime.

Figure 2.9 shows the BDD for function $F =$ before and after simplification. The node count is reduced from 16 to 7. In general, the size of the BDD is quite sensitive to variable order.



(a) Before Simplification.

(b) After Simplification.

Figure 2.9: Logic Simplification.

2.3.4 Decomposition

Often, the circuit generated by an HDL will contain gates that are too complex for direct implementation. These gates must be recursively broken down in to smaller gates, until they can be properly handled by a technology mapper. Usually the requirement is that gates be decomposed into basic gates such as AND, OR, XOR, MUX gates. A number of papers have been published on the subject of [11][22][2][16].

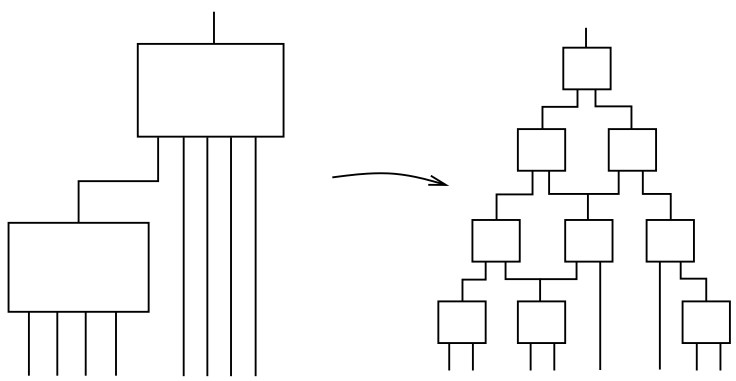


Figure 2.10: Decomposition.

Chapter 3

Sharing Extraction

3.1 Introduction

The purpose of sharing extraction is to extract common functions from among gates in the network to save area. It is similar to decomposition, where large gates get broken down in to smaller gates of lesser complexity, but has the added task of finding sharing opportunities at the same time.

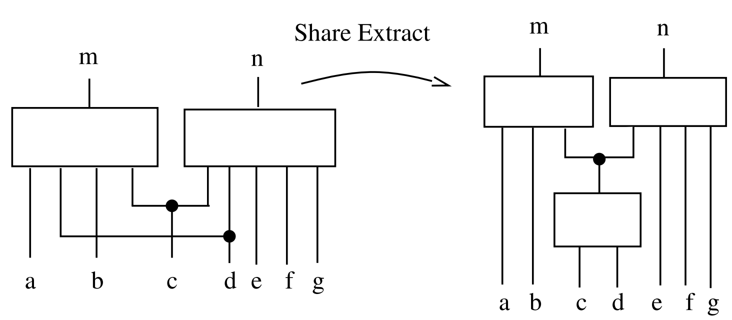


Figure 3.1: Sharing Extraction.

Sharing extraction plays a direct role in the outcome of area quality produced. It is of very practical value, as demonstrated by the highly effective cubeset based kernel extraction algorithm. But while an effective sharing extraction algorithm exists for cubeset based systems, it's treatment in BDD based systems to date has been weak. An effective sharing extraction algorithm is required if BDD based systems are to produce area quality competative with cubeset based systems. In this chapter, we describe a BDD based sharing extraction algorithm that addresses this problem.

3.2 Related Works

Perhaps the most widely used sharing extraction algorithm is the cubeset based kernel extraction described in [3]. 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 factoriztions 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 constrast, *passive* sharing extraction finds sharing only after the fact.

Sawada et. all [18] describe a BDD based equivalent for kernel extraction. While they use BDDs to represent logic functions, they are represented in 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[22] takes an approach to synthesis that moves away from cubesets altogether. They identify good decompositions by relying heavily on structural clues in the BDD. 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. all [13] 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 approach makes several improvements over BDS. First, their sharing extraction step is interleaved with decomposition so that sharing can be found earlier, avoiding redundant computations. Second, they retain don't care information across network transformation to increase flexibility in matching. However, their's is still a passive sharing extraction.

3.3 Overview

Our sharing extraction algorithm shares similarities with the well known kernel extraction algorithm, used in cubeset based systems. Like kernel extraction, our algorithm 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 disjunctive extractors because they can be found efficiently on the BDD. Later the runtime of the algorithm is improved by limiting extractors to two variables, which we shall show, does not adversly affect the area quality produced.

In the second step, common extractors among multiple gates are *selected* for sharing. Committing some extractors destroys 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. The remainder of this chapter will focus on the enumeration and selection of 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. We call the function produced by taking function F and setting it's variable x to the constant 1, the positive cofactor of F with respect to x and is denoted by $F|_x$. Similarily, the function produced by taking function F and setting it's variable x to the constant 0 is called the negative cofactor of F with respect to x, and is denoted by $F|_x$. $[F, C]$ 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.

3.4 Functional Extraction

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

$$F(X) = R(e, X_R) \tag{3.1}$$

$$R(e, X_R) = eR_1(X_R) + \overline{e}R_2(X_R) \tag{3.2}$$

$$e = E(X_E) \tag{3.3}$$

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.

$$R_1 = [F, E] \tag{3.4}$$

$$R_2 = [F, \overline{E}] \tag{3.5}$$

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 [17] 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 [7, 20, 8]. Applying the restrict operator, the final equations for the remainder and extractor are shown below:

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

3.5 Disjunctive 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 disjunctive extractors.

An extractor is *disjunctive* if it does not share support with its remainder. In contrast, an extractor is *conjunctive* if it does share support with its remainder. Examples of disjunctive and conjunctive extraction are shown in Figure 3.2. Disjunctive extractions are ideal for area, because they form a perfect partition of the function, where the remainder and extractor produced have no redundancies between them. However, disjunctive extractors are not always available, in which case, conjunctive extraction is required to break the function down.

(a) Conjunctive Extraction.

(b) Disjunctive Extraction.

Figure 3.2: Conjunctive vs. Disjunctive Extraction.

It is important to note that by limiting the solution space to disjunctive extractors, some sharing opportunities will be missed. Restricting candidate extractors is necessary however, because the generalized sharing extraction problem is NP hard. Nevertheless, disjunctive extractors are good candidates because they can be found and matched quickly. We show experimentally that they are effective in reducing area.

3.5.1 Extractor Types

In addition to being disjunctive, the extractor considered must satisfy additional properties to avoid repeating redundant computations. The first restriction is that extractors be *prime*, that is, an extractor of size N must not be disjunctively extractable by a function of size less than N. For example, *abcd*, can be extracted by *abc*, however, since *abc* can also be extracted by *ab*, *abc* is not considered a valid extractor. The motivation for this restriction is to reduce processing and memory consumption. Without this restriction a function may have on the order of $O(N^N)$ disjunctive extractors, where N is the number of variables of the function. With the restriction, the number of disjunctive extractors is limited to $O(N^2)$. Prime extractors can be shared recursively to find larger, non-prime, extractors.

Another condition that must be satisfied, is that the extractor must evaluate to 0 when the input vector is $\mathbf{0}$. An extractor and its complement produce the same extraction; computing both is redundant. The condition ensures that only one polarity of the extractor is considered. Forcing extractors to satisfy $E(\mathbf{0}) = 0$, ensures that $\overline{E}$ is not considered, since $\overline{E}(\mathbf{0}) = 1$.

In total, three properties must be satisfied for an extractor of size N to be valid.

Condition 1: The extractor forms a disjunctive extraction.

Condition 2: The extractor cannot be extracted by a function of less than N variables.

Condition 3: The extractor evaluates to '0' when all inputs are false.

3.5.2 Enumerating Extractors

The enumeration step identifies all disjunctive extractors for every gate in the Boolean network. Once enumerated, the extractors are matched to find extractors that are shared by multiple gates. The enumeration algorithm works by considering all combinations of variables. For each combination of variables, the algorithm determines whether the set of variables can form a disjunctive extractor. An example of a function and its valid extractors is shown below.

Example 1 *The valid extractors of* $F = abc + d + e$ *are* $\Phi(F) = \{ab, bc, ac, d + e\}$.

The disjunctive extractors in Example 2 are more difficult to find by inspection.

Example 2 *The valid extractors of* $F = \overline{a}\,b\overline{e} + a\overline{b}\,\overline{e} + \overline{a}b\overline{f} + a\overline{b}\,\overline{f} + cef$ *are* $\Phi(F) = \{a \oplus b, ef\}$

It turns out, $E = a \oplus b$ is a disjunctive extractor with $R = E\overline{e} + E\overline{f} + cef$ as the corresponding remainder. And $E = ef$ is a disjunctive extractor with $R = \overline{a}b\overline{E} + cE$ as the corresponding remainder. While identifying extractors by inspection may seem difficult, there is a relatively efficient algorithm to find disjunctive extractors on the BDD. The answer may be efficiently computed on the BDD by checking for equivalence between certain cofactors of F.

Theorem 1 *Let E be an N variable disjunctive extractor of F. Let $S = \{S_0, \cdots, S_{2^N-1}\}$ be the set of all minterms of E. Then E is a disjunctive extractor of F iff all cofactors of F with respect to the minterms in S map to exactly two functions (R_1 and R_2).*

CASE: $\Leftarrow$

1.

$$F = S_0 \cdot F|_{S_0} + \cdots + S_{2^N-1} \cdot F|_{S_{2^N-1}} \qquad \text{By Shannon's expansion}$$

Let $U = \{U_0, \cdots, U_{J-1}\}$ be the minterms of S such that $F|_{U_i} = R_1$, $0 \leq i \leq J-1$.

Let $V = \{V_0, \cdots, V_{K-1}\}$ be the minterms of S such that $F|_{V_i} = R_2$, $0 \leq i \leq K-1$.

And U and V form a partition of S; $U \cap V = \oslash$, $U \cup V = S$

$$F = (U_0 + \cdots + U_{J-1}) \cdot R_1 + (V_0 + \cdots + V_{K-1}) \cdot R_2$$

Since U and V form a partition of S, $U_0 + \cdots + U_{J-1} = (V_0 + \cdots + V_{K-1})'$.

A disjunctive extraction is possible by setting $e = U_0 + \cdots + U_{J-1}$ and $F = e \cdot R_1 + \overline{e} \cdot R_2$.

CASE: $\Rightarrow$

2. E is a disjunctive extractor of $F \Rightarrow F$ can be written as $F(X) = H(X_{\overline{E}}, e), e = E(X_E)$, where X is the support of F, X_E is the support of E, and $X_{\overline{E}} = X - X_E$.

$$\begin{aligned} F &= H(X_{\overline{E}}, e) \\ &= e \cdot H(X_{\overline{E}}, e)|_e + \overline{e} \cdot H(X_{\overline{E}}, e)|_{\overline{e}} \qquad \text{By Shannon's expansion} \end{aligned}$$

$$= E(X_E) \cdot H(X_{\overline{E}}, e)|_e + \overline{E}(X_E) \cdot H(X_{\overline{E}}, e)|_{\overline{e}}$$

Let $U = \{U_0, \cdots, U_{J-1}\}$ be the minterms that make up the on-set of $E(X_E)$.

Let $V = \{V_0, \cdots, V_{K-1}\}$ be the minterms that make up the off-set of $E(X_E)$.

$U \cup V$, enumerate all the minterms of variables in X_E.

$$F = (U_0 + \cdots + U_{J-1}) \cdot H(X_{\overline{E}}, e)|_e + (V_0 + \cdots + V_{K-1}) \cdot H(X_{\overline{E}}, e)|_{\overline{e}}$$

Enumerating the cofactors of F with respect to the minterms of X_E, we have, $F|_{U_0} = \cdots = F|_{U_{J-1}} = H(X_{\overline{E}}, e)|_e$ and $F|_{V_0} = \cdots = F|_{V_{K-1}} = H(X_{\overline{E}}, e)|_{\overline{e}}$. The cofactors of F with respect to the minterms of E map to exactly two functions.

3. Q.E.D.

The cofactor condition can be checked fairly quickly. Cofactors with respect to a cube can be determined in $O(|G|)$ time. And cofactors can be compared, on the BDD, in constant time. For fixed N, the overall complexity of determining if a set of variables can be disjunctively extracted is $O(|G|)$.

The algorithm for determining if a set of variables form a valid, disjunctive extractor works as follows: All 2^N cofactors of F with respect to the N variables considered for extraction, are computed. If all cofactors map to exactly two functions, and the extractor satisfies the three conditions for valid extractors, then the set of variables form a valid, disjunctive extractor. Condition 1 is met by construction. To meet Condition 2, extractors of smaller size are enumerated before extractors of larger size. Variables that belong to extractors of smaller size will not be considered when finding extractors of larger size. For example, all two variable extractors are found first. Those variables that belong to a two variable extractor are not considered when enumerating three variable extractors. This ensures no three variable extractor contains a two variable extractor. Finally, functions are inverted if they do not meet the 3rd condition.

Once a set of variables X_E is determined to have a valid, disjunctive extraction, the exact function to be extracted is computed. The cofactors of F with respect to the minterms of X_E, map to exactly two functions, R_1 and R_2. The extractor is computed by setting the on-set of the extractor to be the sum of those minterms c where $F|_c = R_1$. It follows that the off-set is composed of those minterms c where $F|_c = R_2$. The extractor is then complemented, if required, to satisfy the third condition of valid extractors, $E(\mathbf{0}) = 0$. The algorithm to find disjunctive extractors is shown in Figure 1.

The enumeration algorithm is now applied to the difficult function in Example 2. For $X_E = \{a, b\}$, the four cofactors of F with respect to these two variables are listed below:

$$F|_{\overline{a}\overline{b}} = cef \tag{3.6}$$

$$F|_{\overline{a}b} = \overline{e} + \overline{f} + cef \tag{3.7}$$

$$F|_{a\overline{b}} = \overline{e} + \overline{f} + cef \tag{3.8}$$

$$F|_{ab} = cef \tag{3.9}$$

$$\tag{3.10}$$

All cofactors map to exactly two functions, so we know from Theorem 1 that $\{a, b\}$ can be extracted disjunctively. The two cofactors are $R_1 = \overline{e} + \overline{f} + cef$ and $R_2 = cef$. Two minterms produce cofactor R_1. They are the cofactors with respect to minterms $\overline{a}b$ and $a\overline{b}$. These two minterms form the on-set of the extractor. Hence the extractor is $E = a \oplus b$. The Remainder is $R = \overline{E}(cef) + E(\overline{e} + \overline{f} + cef)$. Computing the cofactors with respect to variables $\{e, f\}$ determines that ef is also a disjunctive extractor. All other combinations of variables result in a mapping to more than two cofactors of F, and hence those variable combinations do not produce disjunctive extractors.

3.5.3 Matching Extractors

Once the extractors are enumerated, the extractors are matched to determine if there is sharing. The candidate extractors are matched by computing an integer signature for the extractors. The signature is a bit vector, where each bit represents a minterm of the extractor. The bit is a '1' if the minterm belongs to the on-set of the extractor, '0' if

Algorithm 1 *Finding Disjunctive Extractors*

```
findExtractors( F ) {
  L = ∅;
  forall( variable combinations X_E of F ) {
    R_1 = ∅;
    R_2 = ∅;
    E = '0';
    isDisjunctive = TRUE;
    forall( minterms C of X_E ) {
      if( R_1 ≠ F|_C ∧ R_2 ≠ F|_C ) {
        if( R_1 = ∅ )
          R_1 = F|_C;
        else if( R_2 = ∅ )
          R_2 = F|_C;
        else{
          isDisjunctive = FALSE;
          break;
        }

        if( R_2 = F|_C )
          E = Or(E, C);
      }
      if( isDisjunctive = FALSE )
        break;
    }
    if( E(0) = 1 )
      E = ¬E;

    L = L + E;
  }
  return L ;
}
```

it belongs to the off-set. Because extractors are limited to support of size 5, at most 32 bits are required to represent a function. Comparing extractors for equivalence amounts to comparing the signatures (an integer comparison) and the support of the extractors.

The value of the signature depends on the ordering of the support variables. For example, the signature for function $F = abc+d$, with variable order {a,b,c,d} is *1110101010101010*, while the signature for F with variable order {d,c,b,a} is *1111111110000000*. To ensure that the variable orders are consistent, the support is sorted by the address location of the variables before the signature is computed.

Each extractor is rated by a cost function based on the size of the extractor and number of instances in the circuit. At each step, the extractor with the lowest cost is accepted in a greedy fashion. The cost function is computed as $C = N - (N - 1) * M$, where N is the support size and M is the number of matches found.

3.5.4 Analysis

The sharing extraction algorithm just described presents a novel BDD based method for finding extractors actively; a task that previously was not available for BDD based systems. It differs from traditional sharing extraction techniques in two major ways. First it allows the entire synthesis flow to be BDD based. The compactness and efficiency with which Boolean operation can be performed on the BDD, ultimately translates into faster runtimes than could be achieved with cube sets. Second, this algorithm is able to extract Boolean gates such as XOR gates, which was difficult to do on cube sets.

Although the algorithm described is well suited for BDDs, it can not be as efficiently performed on cube sets. While computing cofactors on the cube sets is also a linear time operation with respect to the size of the logic function, there is difficulty in comparing cofactors for equivalence. Unlike in the BDD, the cube set representation of logic functions are not canonical and comparison of logic functions cannot be performed in constant time. Even if this difficulty could be overcome, the runtime benefits of using

BDDs would be lost when performing the sharing extraction technique on cube sets.

While the sharing extraction algorithm described can theoretically handle extractors of arbitrary size, for practical purposes, the size of extractors considered must be restricted. Large extractors present a difficulty in computational complexity because the number of variable combinations that need to be considered grows exponentially with the size of the extractor. There are $O(N^M/M^M)$ variable combinations to consider for extractors of size M and functions of size N. The complexity becomes unmanageable fairly quickly for values of M greater than 5. In the next section we show that the algorithm works well for small values of M and give further improvements to reduce runtime.

3.6 Disjunctive Two Variable Extraction

Sharing Extraction with large extractors is slow because of the exponential growth in the number of variable combinations that need to be considered. In this section, the extractor size is limited to two variables. In addition to reducing the number of variable combinations considered for extraction, two variables extractors have some added properties that make their extraction incremental and fast. It will be shown experimentally that typical circuit designs are dominated by two variable extractors and only a negligible area penalty is experienced when ignoring large extractors.

Definition 1 *Given function F, extractor E and remainder R, good extractors are two variable extractors whose variables are disjunctive from R.*

3.6.1 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:

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

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.

3.6.2 Computing Extraction

Good extractors require equivalence between certain cofactors of F. For now, we show how to compute extraction for a good AND extractor. The same procedure shown here can be applied to derive extractions for all other good extractors.

Theorem 2 *$E = ab$ is a good extractor of F iff $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.*

PROVE: If $E = ab$ is a good extractor of F then $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.

1.

$$\begin{aligned} F &= R_1ab + R_2\overline{ab} \\ &= R_1ab + R_2\overline{a}b + R_2\overline{a}\overline{b} + R_2a\overline{b} \end{aligned}$$

2.

$$\begin{aligned} F|_{\overline{a}\overline{b}} &= R_2 \\ F|_{a\overline{b}} &= R_2 \\ F|_{\overline{a}b} &= R_2 \end{aligned}$$

PROVE: If $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$ then $E = ab$ is a good extractor of F.

3.

$$R_1 = [F, ab], \qquad \text{from Equation 3.4.}$$

$F|_{a\overline{b}}$, $F|_{\overline{a}b}$, $F|_{\overline{a}\overline{b}}$ are don't cares. Set them to zero.

$$R_1 \Rightarrow F|_{ab}$$

R_1 does not contain a or b.

4.

$$\begin{aligned} R_2 &= [F, \overline{ab}] \\ &= [F|_{a\overline{b}}a\overline{b} + F|_{\overline{a}b}\overline{a}b + F_{\overline{a}\overline{b}}\overline{a}\overline{b}, \overline{ab}] \end{aligned}$$

Given $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$,

$$\begin{aligned} R_2 &= [F|_{a\overline{b}}(a\overline{b} + \overline{a}b + \overline{a}\overline{b}), \overline{ab}] \\ &= [F|_{a\overline{b}}\overline{ab}, \overline{ab}] \\ &\Rightarrow F|_{a\overline{b}} \end{aligned}$$

R_2 does not contain a or b.

5.

$$\begin{aligned} R &= eR_1 + \overline{e}R_2 \\ &= eF|_{ab} + \overline{e}F|_{a\overline{b}} \end{aligned}$$

The remainder contains neither a or b.

6. Q.E.D.

Similarly, cofactor conditions for the other four good extractors exist as well. These cofactor conditions are listed in Table 3.1.

Let's compare the runtime of extraction for arbitrary functions and good functions. Determining if function $E(a, b)$ is a good extractor, using the first algorithm, requires an extraction computation for each of the 5 extractor types. Each extraction computation requires 2 restrict operations. That's a total of 10 restrict operations. By comparison, the second algorithm computes four cofactors of F. The cofactors are computed once and

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

Table 3.1: Cofactor conditions for good extraction

reused in the detection of each of the extractor types. On the BDD, with it's recursive cofactoring structure, single cube cofactors are computed in $O(G)$ time, where G is the number of BDD nodes.

The complete extraction search algorithm is shown in algorithm 2. 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 $O(N^2G)$.

3.7 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 3 *Let E_1 and E_2 be arbitrary good extractors of F. $Supp(E_1) = \{a, b\}$, $Supp(E_2) = \{c, d\}$ and $Supp(E_1) \bigcap Supp(E_2) = \emptyset$. If R is the remainder of F extracted by E_1, then E_2 is a good extractor of R.*

Algorithm 2 *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.

}}

NOTE: Here we show this is true for the case where $E_1 = ab$ and $E_2 = a + b$. The same analysis can be applied to show the theorem is true for other combinations of good extractors.

CASE: $E_1 = ab$ and $E_2 = a + b$

1. E_1 is a good AND extractor $\Rightarrow F|_{a\bar{b}} = F|_{\overline{a}b} = F|_{\overline{ab}}$

   E_2 is a good OR extractor $\Rightarrow F|_{c\overline{d}} = F|_{\overline{c}d} = F|_{cd}$

2.

$$\begin{aligned} R|_{cd} &= (eF|_{ab} + \overline{e}F|_{a\bar{b}})|_{cd} \\ &= eF|_{abcd} + \overline{e}F|_{a\bar{b}cd} \end{aligned}$$

3.

$$\begin{aligned} R|_{c\overline{d}} &= (eF|_{ab} + \overline{e}F|_{a\bar{b}})|_{c\overline{d}} \\ &= eF|_{abc\overline{d}} + \overline{e}F|_{a\bar{b}c\overline{d}}, \qquad \text{Using } F_{c\overline{d}} = F_{cd}, \\ &= eF|_{abcd} + \overline{e}F|_{a\bar{b}cd} \end{aligned}$$

4.

$$\begin{aligned} R|_{\overline{c}d} &= (eF|_{ab} + \overline{e}F|_{a\bar{b}})|_{\overline{c}d} \\ &= eF|_{ab\overline{c}d} + \overline{e}F|_{a\bar{b}\overline{c}d} \qquad \text{Using } F_{\overline{c}d} = F_{cd}, \\ &= eF|_{abcd} + \overline{e}F|_{a\bar{b}cd} \end{aligned}$$

5. $R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d} \Rightarrow c + d$ is a good extractor of R.
6. Q.E.D.

Thus we can obtain some good extractors of R by copying them from F. 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 4 *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.*

PROVE: If E is a "copy" extractor or "new e" extractor of R, then E is a good extractor of R.

1. Theorem 3 says "copy" extractors are good extractors. "new e" extractors are good extractors by construction.

PROVE: If E is a good extractor of R then E is a "copy" extractor or "new e" extractor.

NOTE: Again, for compactness, we only prove this for the case where $E_1 = ab$ and $E_2 = c + d$.

2. $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$

   $R = eF|_{ab} + \overline{e}F|_{a\overline{b}}$

   $R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d}$

3.

$$\begin{aligned}
& R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d} \\
& e(F|_{ab})|_{cd} + \overline{e}(F|_{a\overline{b}})|_{cd} \\
& = e(F|_{ab})|_{c\overline{d}} + \overline{e}(F|_{a\overline{b}})|_{c\overline{d}} \\
& = e(F|_{ab})|_{\overline{c}d} + \overline{e}(F|_{a\overline{b}})|_{\overline{c}d} \quad * \\
\Rightarrow\ & (F|_{ab})|_{cd} = (F|_{ab})|_{c\overline{d}} = (F|_{ab})|_{\overline{c}d} \\
& (F|_{a\overline{b}})|_{cd} = (F|_{a\overline{b}})|_{c\overline{d}} = (F|_{a\overline{b}})|_{\overline{c}d} \\
\Rightarrow\ & (F|_{a})|_{cd} = (F|_{a})|_{c\overline{d}} = (F|_{a})|_{\overline{c}d}
\end{aligned}$$

4. From *, we have

$$\begin{aligned}
& (F|_{\overline{a}\overline{b}})|_{cd} = (F|_{\overline{a}\overline{b}})|_{c\overline{d}} = (F|_{\overline{a}\overline{b}})|_{\overline{c}d} \\
& (F|_{\overline{a}b})|_{cd} = (F|_{\overline{a}b})|_{c\overline{d}} = (F|_{\overline{a}b})|_{\overline{c}d} \\
\Rightarrow\ & (F|_{\overline{a}})|_{cd} = (F|_{\overline{a}})|_{c\overline{d}} = (F|_{\overline{a}})|_{\overline{c}d}
\end{aligned}$$

5. $F|_{cd} = F|_{c\overline{d}} = F|_{\overline{c}d}$

   Case: Both c and d are elements of F.

   Then $c + d$ is a good extractor of F so $c + d$ is a "copy" extractor.

   Case: One of c or d is the e variable.

   Then $c + d$ is a "new e" extractor.

6. Therefore, an OR extractor must be either a "copy" 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.

3.8 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 5 *$E_1(a, b)$ 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.*

Here we only consider the case where $E_1(a, b) = ab$ (an AND extractor). The analysis presented can be applied to prove the proposition is true for all functions of a and b.

Function $E_2(b, c)$ can be one of 5 extractor functions.

Case: $E_2 = bc$

E_1 is a good AND extractor $\Rightarrow F|_{a\overline{b}} = F|_{\overline{a}b} = F|_{\overline{a}\overline{b}}$ (1)

E_2 is a good AND extractor $\Rightarrow F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{\overline{b}\overline{c}}$ (2)

1.

$$F|_{a\overline{c}} = (F|_{a\overline{c}})|_b b + (F|_{a\overline{c}})|_{\overline{b}} \overline{b}$$

$$
\begin{aligned}
&= F|_{ab\overline{c}}b + F|_{a\overline{b}\overline{c}}\overline{b} \qquad \text{using (1)} \\
&= F|_{\overline{a}\overline{b}\overline{c}}b + F|_{\overline{a}\overline{b}\overline{c}}\overline{b} \\
&= F|_{\overline{a}\overline{b}\overline{c}}
\end{aligned}
$$

2.

$$
\begin{aligned}
F_{\overline{a}c} &= (F|_{\overline{a}c})|_b b + (F|_{\overline{a}c})|_{\overline{b}}\overline{b} \\
&= F|_{\overline{a}bc}b + F|_{a\overline{b}c}\overline{b}, \qquad \text{using (1)} \\
&= F|_{\overline{a}\overline{b}c}b + F|_{\overline{a}\overline{b}c}\overline{b} \qquad \text{using (2)} \\
&= F|_{\overline{a}\overline{b}\overline{c}}b + F|_{\overline{a}\overline{b}\overline{c}}\overline{b} \\
&= F|_{\overline{a}\overline{b}\overline{c}}
\end{aligned}
$$

3. Similarly,

$$
\begin{aligned}
& F|_{\overline{a}\overline{c}} = F|_{\overline{a}\overline{b}\overline{c}} \\
\Rightarrow\ & F|_{\overline{a}c} = F|_{a\overline{c}} = F|_{\overline{a}\overline{c}} \\
\Rightarrow\ & E_3(a,c) = ac \text{ is a good AND extractor of } F.
\end{aligned}
$$

Similarly, if $E_2 = b\overline{c}$, then $F|_{ab} = F|_{a\overline{b}} = F|_{\overline{a}\overline{b}}$ which implies $E_3(a,c) = a\overline{c}$ is a good extractor of F.

Case: $E_2 = b + c$

Here we show that $b + c$ cannot be a good extractor. Assuming that both ab and $b + c$ are good extractors results in the contradiction that F is independent of a.

E_1 is a good AND extractor $\Rightarrow F|_{a\overline{b}} = F|_{\overline{a}b} = F|_{\overline{a}\overline{b}}$ (1)

E_2 is a good OR extractor $\Rightarrow F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{bc}$ (2)

4.

$$
\begin{aligned}
F|_{\overline{b}c} &= (F|_{\overline{b}c})|_a)a + (F|_{\overline{b}c})|_{\overline{a}})\overline{a} \\
&= F|_{a\overline{b}c}a + F|_{\overline{a}\overline{b}c}\overline{a}, \qquad \text{using (1)} \\
&= F|_{\overline{a}\overline{b}c}a + F|_{\overline{a}\overline{b}c}\overline{a} \\
&= F|_{\overline{a}\overline{b}c} \\
&= (F|_{\overline{b}c})|_{\overline{a}}
\end{aligned}
$$

5.

$$
\begin{aligned}
F|_{\overline{b}c} &= (F|_{\overline{b}c})|_{\overline{a}} \\
&\Rightarrow F|_{\overline{b}c} \text{ is not dependent on } a.
\end{aligned}
$$

6. Since $F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{bc}$, $F|_{b\overline{c}}$ and $F|_{bc}$ also do not depend on a.

7.

$$
\begin{aligned}
F|_{\overline{b}\overline{c}} &= (F|_{\overline{b}\overline{c}})|_{a}a + (F|_{\overline{b}\overline{c}})|_{\overline{a}}\overline{a} \\
&= F|_{a\overline{b}\overline{c}}a + F|_{\overline{a}\overline{b}\overline{c}}\overline{a}, \qquad \text{using (1)} \\
&= F|_{\overline{a}\overline{b}\overline{c}}a + F|_{\overline{a}\overline{b}\overline{c}}\overline{a} \\
&= F|_{\overline{a}\overline{b}\overline{c}} \\
&= (F|_{\overline{b}\overline{c}})|_{\overline{a}} \\
&\Rightarrow F|_{\overline{b}\overline{c}}\text{does not depend on } a
\end{aligned}
$$

8. $F|_{bc}$, $F|_{b\overline{c}}$, $F|_{\overline{b}c}$, $F|_{\overline{b}\overline{c}}$ all do not depend on $a \Rightarrow F$ does not depend on a, a contradiction.

Therefore, $b + c$ cannot be a good extractor of F when ab is a good extractor of F. Similarly, $E_2 = \overline{b}c$ and $E_2 = b \oplus c$ cannot be good extractors of F when $E_1 = ab$ is a good extractor of F.
We have shown that for the two valid functions of E_2 $\exists$ $E_3(a, b)$ that is a good extractor of F.

9. Q.E.D.

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%.

3.9 Summary

In this chapter we described a fast, BDD based, sharing extraction algorithm that aims to reduce area. The first incarnation of this algorithm finds all disjunctive extractors of size, up to five variables. The extractors are found when the cofactors for all permutations of cubes for the extraction variables in the equation, map to exactly two functions. A unique bit string representation of the extractor is computed and hashed to find matching extractors, which can be extracted and shared.

The computational complexity of finding large extractors grows polynomially with the gate size, and exponentially with the extractor size. For extractors of size K, and functions of size N, the complexity of this problem is $O(N^K \cdot 2^K|G|)$. Because of the exponential growth complexity, K is reasonable for only values of less than or equal to five.

Several refinements to the algorithm have been made to reduce computational complexity further. As we will show experimentally, small values of K are sufficient in capturing most sharing opportunities. In fact, of extractors sized two to five, two variable extractors account for 99.7% of shared extractors found. The computational complexity of finding extractors of size two is $O(N^2|G|)$. This is reduced further by finding extractors incrementally. Disjunctive extractors of a function are also disjunctive extractors of the functions produced by extraction. A heuristic, using the transitive property of disjunctive extractors, look for extractors between adjacent variables. In total, the refinements produce an algorithm that finds disjunctive extractors in $O(N|G|)$ time, which is in line with the complexity of other synthesis transformations such as decomposition.

Chapter 4

Folded Logic Transformations

4.1 Introduction

With the rapid shrinking of process geometries and corresponding abundance in logic resources, the complexity of designs has been growing at a rate that is making it a challenge for logic synthesis tools to keep their runtimes manageable. In the design of large circuits, design reuse through hierarchy or repetition of logic structures is often applied to reduce design effort. This theme can be used to speed up synthesis speed. In this chapter we exploit the inherent regularity in logic circuits to share the transformation results between equivalent logic structures, called logic folding, with the focus of improving runtime.

4.2 Related Works

Regularity awareness was first used in placement in an effort to improve circuit density and reduce interconnect by Arikati [1]. Arikati performed regularity extraction by analyzing circuit connectivity and using a signature based approach to recognize regularity. In his algorithm, signatures are used to match the structural relationship between a gate and its neighbors. Regularity extraction starts with slices, with each slice containing a single gate with the same gate type as those in the other slices. Adjacent gates are merged into the slice when their signatures are found to match the signatures of other gates with respect to the other slices. Kutzschebauch [12] later applied this idea to logic synthesis in an effort to reduce the loss of regularity during logic synthesis. While he was able to improve the post synthesis regularity of circuits by on average 57%, the run time, on average, increased by 8%.

4.3 Overview

We use a simpler form of regularity that only considers equivalence between single gates, or pairs of gates. Logic transformations typically operate on one or two gates at a time. When matching single gates, we are interested in whether a gate is functionally equivalent to other gates in the network. When dealing with pairs of gates, we are also interested in how the pair is interconnected. Capturing this regularity information enables us to detect instances in the circuit where the same logic transformation is applied more than once. Noting that many circuits exhibit a fair amount of regularity, and noting that many logic transformations depend solely on the logic functions of the gates, we propose to use logic folding to identify regularities in the circuit, to share the logic transformations and improve runtime.

4.4 Single Gate Matching

Prior to the BDD, equivalence checking between two Boolean functions was expensive. The cube set representation, faced two hurdles when performing equivalence checking. First, cube sets are not canonical and the cost of putting them in a canonical form is expensive. Second, even if cube sets could be made canonical, they still required the comparison of the cubes in order to confirm equivalence. This is much slower than the constant time requirement for equivalence checking with the BDD. Because logical equivalence between gates could not be determined easily, each gate stored it's own copy the logic function, and no attempt to share the logic function representations was made.

In our circuit representation, we take advantage of fast BDD based matching by separating gates from their logic functions. The logic functions are stored in a global function manager where the N variables of a function are mapped to the bottom N generic variables of the function manager. When a new gate is constructed, its logic function is constructed in the global function manager. If the BDD for the logic function finds a match, then the function is shared and the gates are grouped together. Otherwise a new function is added to the global manager.

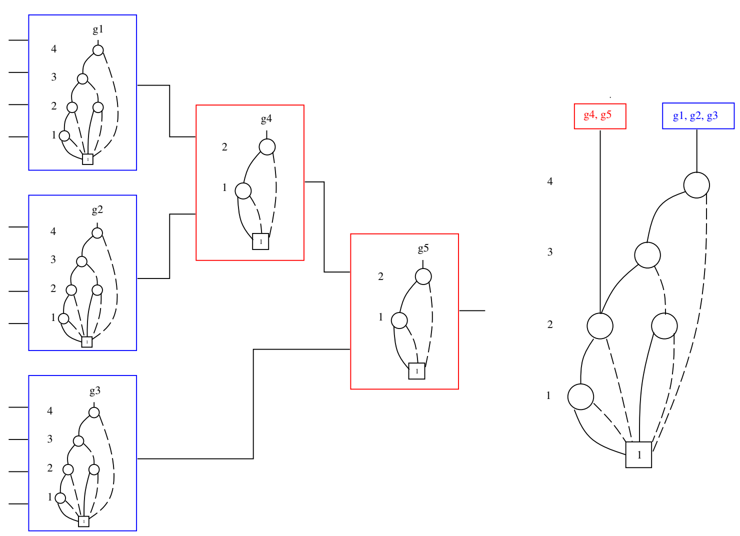


Figure 4.1: Regularity Extraction.

Sharing the BDD function representation among gates in the circuits obviously reduces memory requirements. The runtime advantage, however, is that gates can be grouped by logical equivalence and logic transformations performed on one gate can be shared among all members of the group.

4.4.1 Boolean Matching

Folded logic is not without its problems and limitations. Two Boolean functions can have more than one BDD representation when the variables of the BDDs are mapped differently to the generic variables of the Global BDD Manager. When this occurs, the match is missed and the BDDs are mistaken as different Boolean functions. As a result, logic transformations involving these BDDs must be performed separately. Given that for an n variable function, there are up to $O(n!)$ different variable orderings, it seems unlikely for two equivalent functions to match.

The set of valid variable orders for a function can be dramatically reduced by applying variable reordering. Since BDDs are typically variable ordered anyways, to conserve memory, this comes at no extra cost. However, variable ordering alone is not sufficient for discovering most matches. To illustrate this point, the sifting variable ordering algorithm [15] is applied to several logically equivalent BDDs, whose starting variable orders are randomly chosen. This procedure is applied to the largest gate in each of the MCNC benchmarks. Table 4.1 (PMSIFT) shows the rate of matching obtained. Ideally, matching should be 100% because the logic functions compared are equivalent. However, on average, sifting can only discover 17% of the matches.

The problem described above, called permutation-independent Boolean Matching, has been investigated in the context of library cell binding and logic verification. Ercolani and De Micheli [10] propose a matching algorithm for EPGAs where BDDs are constructed for all possible input permutations of the uncommitted EPGA module, and stored together in a global manager. While practical for EPGA mapping, where logic functions are only compared against the EPGA module, the memory requirements of this approach are not practical when arbitrarily many gates are compared with each other. Debnath and Sasao [9] devise a permutation-independent, canonical form for the logic function where the rows of the truth table are represented as bits in a bit vector. Each bit indicates whether the row is part of the on-set of the function. The size of the bit vector grows exponentially with the size of the support set, and is not practical for regularity detection, where large gates exist. Ciric and Sechen [5] propose a canonical form where the function is represented as the concatenation of minterms. Their algorithm performs an exhaustive branch and bound search for the unique identifier which can be obtained through minterm reordering and variable reordering. Their algorithm also cannot handle the large gates that may exist in a logic network.

Mohnke, Molitor and Malik [14] propose a solution to the Boolean matching problem that does not suffer from the runtime and memory limitations of the algorithms described earlier. Their technique is based on computing signatures for the inputs of the logic functions that are independent of the variable order. The inputs can be sorted by their signatures to generate a variable order. Two BDDs that represent the same Boolean functions will produce the same input signatures. If each input signature is unique, then a unique variable ordering can be created from the signatures, and the resulting BDDs will be equivalent.

One example of an input signature is the *Cofactor satisfy count signature*[14]. For an input variable x, its input signature is defined to be the number of input assignments for which the logic function is true when x is true. In BDD representation, this corresponds to the number of paths to the one terminal and can be computed in $O(|G|)$ time, where $|G|$ is the number of nodes in the BDD. The limitation of this approach is that, unlike the Boolean matching techniques described earlier, it does not result in a canonical form. Input signatures may alias, resulting in non-unique variable orders. In folded synthesis, where the goal is to decrease runtime, some missed matches can be tolerated. Using the Cofactor satisfy count signature to create an initial variable order allows 78% of the matches to be found. The results are shown in Table 4.1 (PMSIG). Mohnke et al report in [14] that for the set of signatures they implement, unique signatures are obtained in 92% of the circuits in the LGSynth91 and ESPRESSO benchmarks.

In spite of the problems described above, the potential benefit of logic folding is huge.

Table 4.1: Matching Rate for Sifting and Input Signature Methods.

Circuit	PMSIFT (%)	PMSIG (%)
C1355.blif	10.42	100.00
C1908.blif	1.14	100.00
C2670.blif	1.00	100.00
C3540.blif	22.68	76.16
C5315.blif	2.38	100.00
C6288.blif	50.72	100.00
C7552.blif	1.00	100.00
dalu.blif	1.06	37.56
des.blif	1.54	26.78
frg2.blif	1.00	25.94
i10.blif	1.00	25.98
i8.blif	1.04	100.00
i9.blif	11.04	100.00
k2.blif	32.74	100.00
pair.blif	79.58	76.36
rot.blif	1.00	100.00
t481.blif	1.02	100.00
vda.blif	100.00	100.00
x3.blif	1.02	17.52
AVERAGE	16.91	78.23

If a match is found early on, there are savings on the immediate logic transformation, as well as on all downstream logic transformations. Folding is essentially free. The cost of folding is to copy BDDs to and from the global BDD manager, but this copying is required anyways when isolating a BDD for variable reordering.

4.5 Matching Gates Pairs

Earlier we showed how a logic transformations result, when applied to a single gate, can be shared among all logically equivalent gates. There are some logic transformations however that work on two gates at a time. For example, elimination collapses one gate into its fanout. In this section we describe how sharing transformations can be extend to pairs of gates.

Two gate pairs P_1 and P_2, have the same logic transformation result when the gate pairs meet two requirements. First, the Boolean function for each of the two gates in P_1 must match with the corresponding gates in P_2. (i.e. $P_1.gate1.bool = P_2.gate1.bool$ and $P_1.gate2.bool = P_2.gate2.bool$). With the Boolean functions of the gates already matched in the shared function manager (described earlier) this problem is easily solved by using a hash table with the two Boolean functions of gate pair as the hash key. Secondly, we need to match how the gates of a gate pair are interconnected. In particular, we need to identify which variables are shared, and the positions that the shared variables take in the support sets. This information is called support configuration. When the Boolean functions and support configurations of two gate pairs match, their transformation results will be the same.

4.5.1 Support Configurations

Support configuration tells us how variables are shared between the two gates. It does not record information about where the support comes from, but rather what position that shared variable takes in the support sets. Therefore, two gate pairs can have very different support sets but identical support configurations. Before explaining how support configuration is computed, we make a few assumptions that are required of the gates. First, no gate has repeating input variables in its support. And second, no gate has constant values in its support. Both conditions can be met by sweeping the circuit for these instances, and simplifying a gate whenever repeated or constant variables are found in its support.

Let S_1 and S_2 be two support sets. A support configuration is an unordered set of pairs where each pair corresponds to a shared variable. The first element of each pair represents the position of the shared variable in S_1 and the second element of each pair represents the position of the shared variable in S_2. The support configuration can be computed in linear time with respect to the size of the support sets.

Example 3 *Let* $S_1 = \{a, d, b, c\}$ *and* $S_2 = \{c, d, e, f, g\}$ *. The arrangement is shown in Figure 4.2. Their support configuration is* $C(S_1, S_2) = \{(1, 1), (3, 0)\}$.

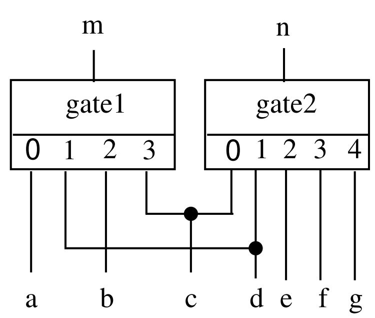


Figure 4.2: Support Configuration Example.

The purpose of computing support configurations is to find matching with support configurations in other gate pairs. Support configuration matching is performed very frequently. Whenever a gate pair is created, its support configuration must be compared with all other existing gates pairs for equivalence. A simple way to compare two support configurations is to do a linear traversal of their lists. However, this is a significantly slower than the constant time, pointer comparison done with Boolean matching.

4.5.2 Characteristic Function

We present a faster way to compare support configurations by computing a characteristic function. In our formulation of the characteristic function, we use BDDs to represent the elements of a set. An element is represented as a Boolean function of $log_2(N)$ variables, where N is the number of elements in the set. Let $x_0, \cdots, x_{K-1}$, be the K variables of the characteristic functions. Then the elements of the set are assigned as follows:

$$
\begin{aligned}
P(0, X) &= x_{K-1} \cdots x_1 \cdot x_0 && (4.1)\\
P(1, X) &= x_{K-1} \cdots x_1 \cdot \overline{x_0} && (4.2)\\
P(2, X) &= x_{K-1} \cdots \overline{x_1} \cdot x_0 && (4.3)\\
P(3, X) &= x_{K-1} \cdots \overline{x_1} \cdot \overline{x_0} && (4.4)\\
& \quad etc \cdots && (4.5)
\end{aligned}
$$

$P(i, X)$ is used to denote the characteristic function for ith element using the variables $X = x_0, \cdots, x_{K-1}$. This representation grows logarithmically with the size of the set, and each element is represented by a single cube. The characteristic functions for the elements are combined to form a support configuration characteristic function.

Let S_1 be a support set of size $|S_1|$. Let S_2 be a support set of size $|S_2|$. Let $C(S_1, S_2) = \{(x_1, y_1), (x_2, y_2), \cdots, (x_K, y_K)\}$ be their support configuration, where K is the number of shared variables. Let X be a set of $log_2(|S_1|)$ variables. Let Y be a set of $log_2(|S_2|)$ variables (independent of X).

Then the support configuration characteristic function is computed as,

$Q = P(x_1, X)P(y_1, Y) + P(x_2, X)P(y_2, Y) + \cdots + P(x_{K-1}, X)P(y_{K-1}, Y)$

The memory requirements for this representation are quite modest; the number of variables of the characteristic function is $log_2(|S_1|)+log_2(|S_2|)$. The major advantage with the characteristic function representation, however, comes from that fact that when stored as a BDD, equivalence between support configurations can be confirmed in constant time.

4.6 Applications

All transformations are characterized by one or two logic functions as input, one or more logic functions for output and a mapping between the variables of the new logic functions and the old. In this section, we give a description of how the folded synthesis technique is applied to speed up the runtime for four logic synthesis transformations: simplification, decomposition, elimination and sharing extraction.

4.6.1 Folded Simplification

A logic expression can be expressed in a number of ways, with some expressions being more compact than others. The goal of simplification is to minimize the complexity of a logic function in an effort to reduce area. In BDD based logic synthesis, one measure of the complexity is the size of its BDD. This size is very sensitive to the variable order chosen and many techniques have been devised to select a variable order that minimizes the node count. BDD based simplification amounts to applying variable reordering on a logic function of a gate, and remapping its support set accordingly. Using the property that two logically equivalent gates have the same result after simplification, simplification is performed on one logic function and the result applied to all instances of that function.

4.6.2 Folded Decomposition

The folded decomposition algorithm is shown in Algorithm 3. Each decomposition is performed one BddClass at a time (and potentially more than one BddInstance at a time). The BDD for the BddClass is decomposed into two or more smaller BDDs. If these BDDs are not found in the Global BDD Manager, new BddClasses are created for them. Otherwise, the existing BddClasses are used. The BddInstances are updated to reflect the changes. If the new BddClasses can be decomposed further, they are added to the heap. The heap is used to decompose the BddClasses in order of non-increasing size of their support set. Decomposing BDD's in this order ensures that no decompositions are repeated.

The folded decomposition process is illustrated in Figure 4.3. The Boolean functions for the gates are stored in a global manager where two unique Boolean functions have been identified (Figure 4.3b). The BDD with the largest support set, cell_i1, is decomposed

Algorithm 3 *Folded Decomposition*

```
DecomposeAll() {
  forall( BddClasses in the Boolean network )
    Push(heap, BddClass, BddClass.numVariables);
  while( f = Pop(heap) ) {
    ⟨g, h, op⟩ = decompose(f);
    if( !bddExists(g) )
      create BddClass for g;
    if( !bddExists(h) )
      create BddClass for h;
    update instances(f, g, h, op);
    if( g has more than two nodes )
      Push(heap, g, g.numVariables);
    if( h has more than two nodes )
      Push(heap, h, h.numVariables);
  }
}
```

first. The decomposition result transforms cell_i1 into the XOR of a XNOR and NOT gate (Figure 4.3c). cell_i1's BDD is then removed from the Global manager and replaced with XNOR, NOR and NOT BDDs (Figure 4.3e). The two original cell_i1 instances are replaced with NOR, XNOR and NOT instances (Figure 4.3d).

4.6.3 Folded Elimination

Elimination is the process of merging nodes on the Boolean network with the goal of removing inter-gate redundancies. An adjacent pair of gates form an elimination pair $\langle G1, G2, pos \rangle$, which consists of a parent gate *G1*, child gate *G2* and a position *pos*. The child gate has fanout to the parent gate and *pos* is the position of the variable in the parent gate that is to be substituted by the child function.

Two elimination pairs, P_1 and P_2, produce the same elimination result if the logic functions of its gates are the same, the position where they connect is the same, and their

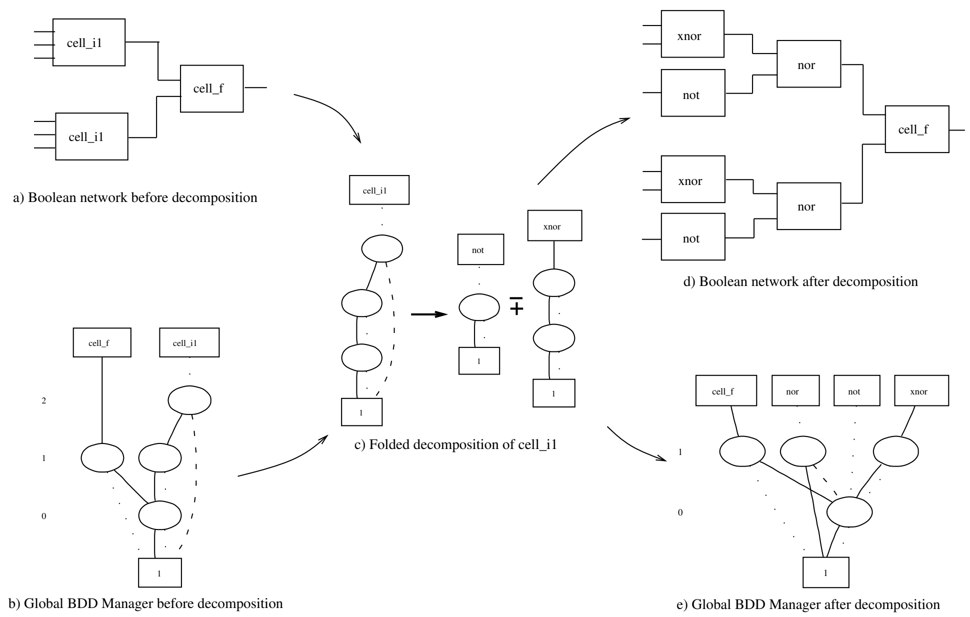


Figure 4.3: Folded Decomposition.

support configurations are the same. i.e. $P_1.G_1 = P_2.G_1$, $P_1.G_2 = P_2.G_2$, $P_1.pos = P_2.pos$ and $C(P_1.G_1, P_1.G_2) = C(P_2.G_1, P_2.G_2)$. A hash table is used to identify elimination pairs with the same gate functions, position *pos* and support configuration. When an elimination pair is created, it is matched against the hash table. Eliminations pairs that match are grouped together. The elimination result can be computed once and shared among all members of the group.

4.6.4 Folded Sharing Extraction

Sharing extraction has two separate computations that can take advantage of regularity. The first computation is the enumeration of extractors. Equivalent functions will produce the same list of disjunctive extractors which can be shared by all instances of the function. This is the cost of computing cofactors between all adjacent variables of the function, to determine if they can be disjunctively extracted. The extractors found are written in terms of generic variables, not in terms of absolute support. This computation is done only once, then the extractor list is enumerated in terms of absolute support for each instance of the function.

Consider the functions $F = ab+cd$ and $G = lm+ad$. In terms of generic variables, the logic functions are identical, $H = x_0x_1+x_2x_3$. The extractors are enumerated on the logic function to produce the following extractors (x_0x_1, x_2x_3). At this point, the expensive process of computing the cofactor conditions has been completed. The extractors for F and G are then enumerated by remapping the generic variables to actual support. F has extractors ab, cd and G has extractors lm, ad.

The second computation where regularity can be taken advantage of, is the computation of remainders. When an extractor is selected for sharing, it must be extracted from it's parent function to produce a remainder. This requires the expensive process of computing the extractor, and then simplifying the function through variable reordering. For an extraction that breaks F down into remainder R and extractor E, extractions that involve the same F and use the same relative positions of the variables of E in F, produce the same remainders. In the example given earlier, the remainder for F when extracting (a, b) is the same as the remainder for G when extracting (l, m) because the logic functions for F and G are the same, and the relative position of the variables of their extractors are the same (using variables (x_0, x_1)). Thus the remainder $R = e + v_2v_3$ is computed once only, and shared by both instances F and G.

4.7 Summary

In this chapter, the regularity of circuits is used to improve runtime by sharing the results of logically equivalent transformations. At the foundation, the logic functions are stored in a common repository where they are shared by the gates that implement them. The fast equivalence checking of BDDs makes this capability possible. In contrast, the Sum of Product representation is not canonical, and the cost of equivalence checking prohibitively expensive. Some transformations, such as decomposition and simplification, depend solely on one logic function. For these transformations, a logic transformation is applied by logic function, and the result applied to all gates that implement it. Other transformations, such as elimination, depend on a pair of gates, and their interconnection (called support configuration). The pair of gates are matched in the way single gates are matched, by comparing their BDD pointers. The support configurations are matched by comparing their canonical, BDD based, characteristic functions. When these transformations involve the same pair of gates and same support configuration, then they share the same transformation result.

Chapter 5

Experimental Results

5.1 FBDD Package

To evaluate our proposal we implemented a complete logic synthesis system, called FBDD, that includes the sharing extraction algorithm and regularity aware techniques described in this thesis. FBDD is a BDD-based, combinational circuit optimization program. It starts with a gate level description of a circuit in the Berkeley Logic Interchange Format (BLIF) [19]. FBDD then applies a set of algorithms to minimize area while also breaking the circuit down into basic gates in preparation for technology mapping. The output produced, is an area optimized, technology-independent circuit in BLIF or structural Verilog format. BLIF output enables a path from FBDD to academic, standard cell or FPGA technology mappers. Industry standard Verilog output allows for integration with a wider array of tools, including commercial tools.

In addition to the new optimization techniques described in this thesis, FBDD also implements many of the standard tasks required of typical synthesis systems. This includes the sweep, elimination, simplification and decomposition steps. Sweep simplifies support sets with repeat inputs or constant inputs. Elimination attempts to remove redundancy between gates. And simplification reduces the complexity of the logic function within gates. Decomposition consists of a set of algorithms to decompose large gates, while minimizing area. The set of Decomposition algorithms we implement are the disjunctive AND, OR and XOR decompositions based on 1,0 and X dominators in BDS. We also implement the Boolean AND/OR decomposition based on their generalized dominator, and variable and functional MUX decomposition [22]. When a decomposition problem is encountered, all algorithms are tried and the solution that produces the lowest BDD node count is selected.

The *sharing extraction* and *decomposition* steps are interleaved. Sharing extraction is applied first, to find as much disjunctive sharing as possible. When no more extractors can be found, decomposition is applied to further break down those gates. As decompositions are applied, new disjunctive extractors may surface so sharing extraction is re-applied. This cycle continues until the circuit is completely decomposed.

FBDD contains over 26000 lines of code, written in the C programming language. Low level BDD storage and manipulation are handled with the CUDD package [21], developed by Fabio Somenzi at the University of Colorado at Boulder. FBDD runs on both the Linux and Solaris operating systems.

5.2 Evaluation Methodology

The experiments are conducted on a dual processor Solaris Blade 1000 with 2.5 GB memory running SunOS version 5.8. Two sets of publically available benchmarks are used in the experiments. The MCNC91 benchmarks [23], are highly reported in academic publications and we use it enable comparison with other works. We use only the combinational, multi-level examples with approximate gate counts of 500 or more for testing. Even so, the circuits obtained from MCNC91 are relatively small by today's standards. To complement them, we also report results on the ITC99 benchmarks [6], which include a set of large processor cores.

In our studies we also use a synthetic benchmark which is made up of repeating instances of a template circuit. For the template circuit we use *rot.blif* from the MCNC91 benchmark. This benchmark is used to study runtime growth with respect to regularity while the keeping the logic content the same. We also run tests on adder circuits of varying bit widths to determine the tools ability to find sharing and handle XOR gates.

When reporting the area of synthesized circuits, the areas of the individual gates in the circuit, after technology mapping, are summed together. The SIS Mapper is used to perform technology mapping to the *lib2.genlib* standard cell library from the MCNC benchmark. Another common measure of area is literal count, however, we use the area after technology mapping as our metric because the tools (FBDD, SIS, BDS) target different levels of decomposition, which has an effect on literal count.

5.3 Experiment Overview

The goal of the experiments is to quantify the area and runtime benefits of our new sharing extraction algorithm and folded synthesis approach. Section 5.4 focuses on our new sharing extraction algorithm. To justify our restriction to only extractors of two variables, we analyze the computational effort required to find extractors of various sizes, in Section 5.4.1, and compare that to their area improvement. In Section 5.4.2 we compare exhaustive, all pairs, two variable, extractor enumeration to fast adjacent pairs of variables enumeration. Finally, a comparison of FBDD with sharing extraction versus FBDD without sharing extraction is given in Section 5.4.3.

Section 5.5 focuses on the scalability of our folded synthesis approach. The folded approach shares logic transformations to reduce runtime. But the effectiveness of this method depends on the proportion of sharable to non-sharable costs. In Section 5.5.1, we use a controlled experiment, where the regularity of the benchmark is increased, while the logic content is held constant. A small, relative growth in the runtime indicates that the sharable costs, make up the majority. Section 5.5.2 we investigate the effectiveness of folded synthesis on the MCNC benchmarks. We report the number of transformations performed, and the number of transformations that could be shared. We do this for each of the major synthesis stages - decomposition, elimination and sharing extraction.

Finally, in Section 5.6 we compare FBDD with two other publically available synthesis systems, SIS and BDS. We compare the runtime and area results against the MCNC and ITC benchmarks in Section 5.6.1 and Section 5.6.2 respectively. The scalability of the tools is evaluated against the regularity controlled, synthetic benchmark in Section 5.6.3. We show an adder example in Section 5.6.4, where sharing extraction and XOR extraction are important.

5.4 Sharing Extraction

5.4.1 Maximum Extractor Size

The runtime of sharing extraction grows exponentially with the size of the extractors considered. We stated that the runtime could be improved by limiting extractors to two variables without much sacrifice in area. In this section we give empirical evidence to support that claim.

The large examples of the MCNC benchmark were synthesized using varying maximum extractor sizes of two to five. The exact algorithm, which enumerates *all* variable combinations, is used. The runtimes spent on sharing extraction are summed together and shown in Figure 5.1. From the figure, a steep trade off between maximum extractor size and runtime can be seen. Sharing extraction with extractors of five variables is over nine times slower than with extractors of two variables. With such large runtimes, sharing extraction dominates the overall runtime of synthesis.

To determine the effect that maximum extractor size has on area, we performed logic synthesis with a maximum extractor size of 5, and collected information on the

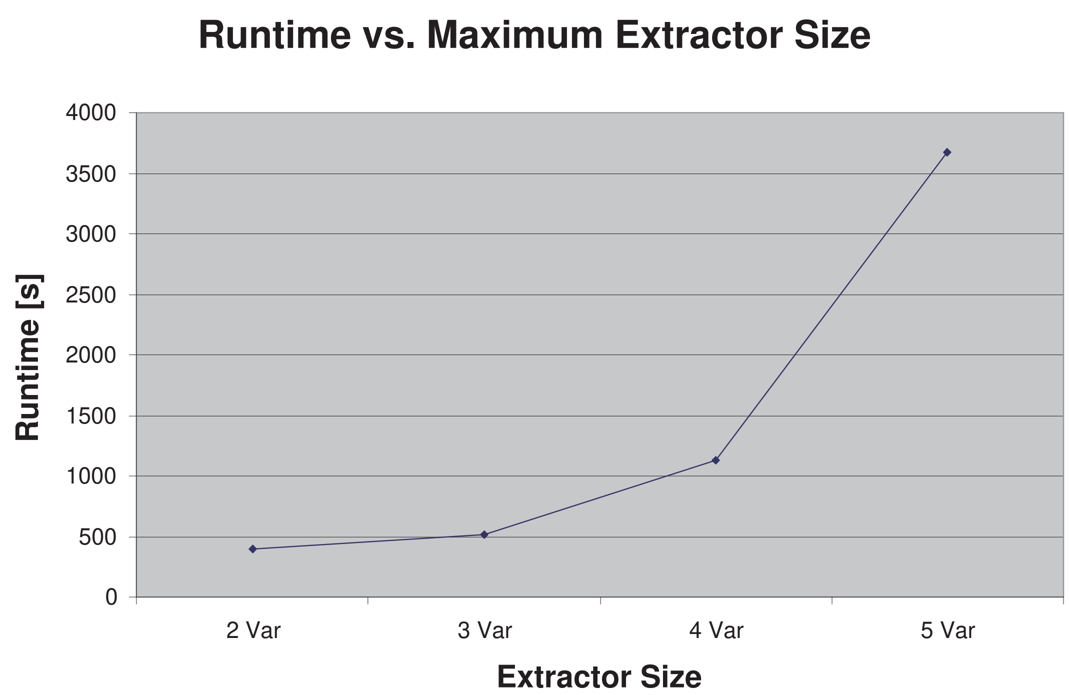


Figure 5.1: Runtime vs. Maximum Extractor Size.

distribution of extractor sizes found. The number of shared extractors found for circuits in the MCNC benchmark are collected and reported in Table 5.1. Each number in the table indicates the number of *prime* extractors found, for a given size. An extractor of size K is *prime* if it cannot be disjunctively extracted by a function of size less than K.

The data shows that two variable extractors clearly make up the majority. On average, extractors of size three through five make up only 0.33% of the shared extractors found, with two variable extractions making up the rest. This is not an entirely obvious result. The number of prime extractors with N variables grows super exponentially with respect to N. As analyzed earlier, there are 5 two variable valid extractors (prime extractors of positive polarity). This number grows to 52 three variable valid extractors and 28620 four variable valid extractors. If circuits were composed of random circuits, the proportion of two variable extractors would be much less. In practice, circuits are typically composed of highly structured logic, such as AND, OR and XOR gates, which can be disjunctively

Table 5.1: Distribution of Shared Extractor Sizes.

Circuit	2 Var	3 Var	4 Var	5 Var
C1355	12	0	0	0
C1908	176	0	0	0
C2670	163	9	0	1
C3540	198	0	0	0
C5315	502	2	3	0
C6288	0	0	0	0
C7552	492	2	0	0
alu4	70	7	4	0
dalu	707	10	0	0
des	1697	0	0	0
frg2	619	0	0	0
i10	785	0	0	0
i8	2096	0	0	0
i9	566	0	0	0
k2	1911	0	0	0
pair	116	6	0	0
rot	73	3	0	0
t481	2733	0	0	0
too_large	359	0	0	0
vda	888	0	0	0
x3	234	0	0	0
Total	14397	39	7	1

extracted using two variable extractors.

The area results produced using the varying maximum extractor sizes are shown in Figure 5.2. With relatively few large extractors available for sharing, the effort put into their detection has little effect on area results. Since the computational cost of finding large extractors is high, and the area gain almost non-existent, the runtime of sharing extraction can safely be improved by ignoring large extractors, without significantly affecting area.

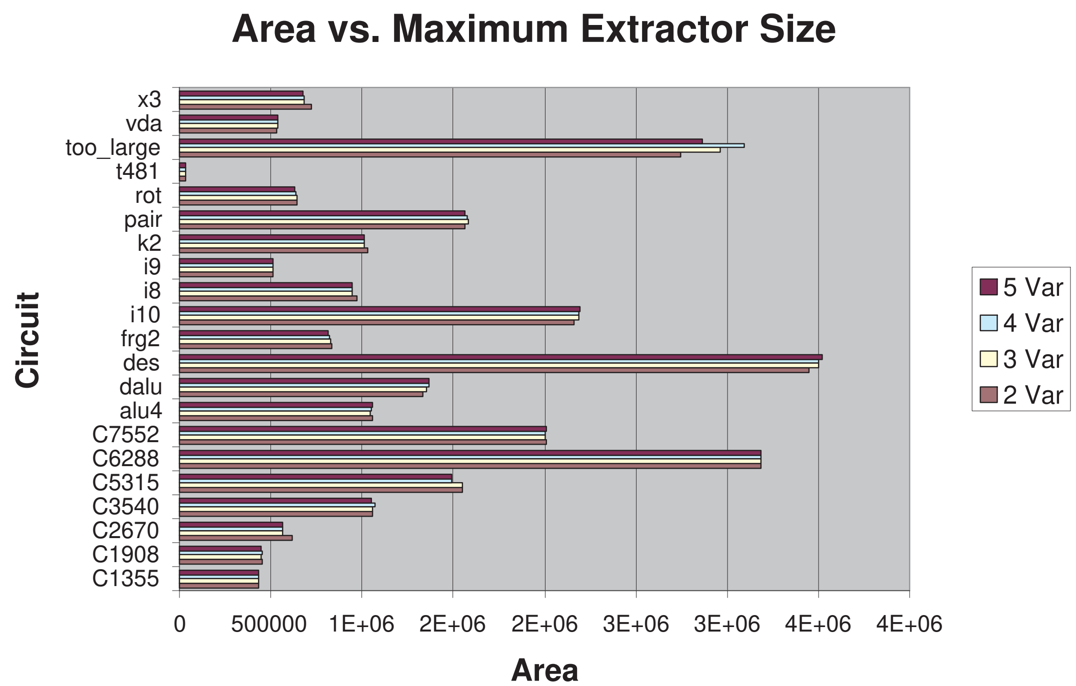


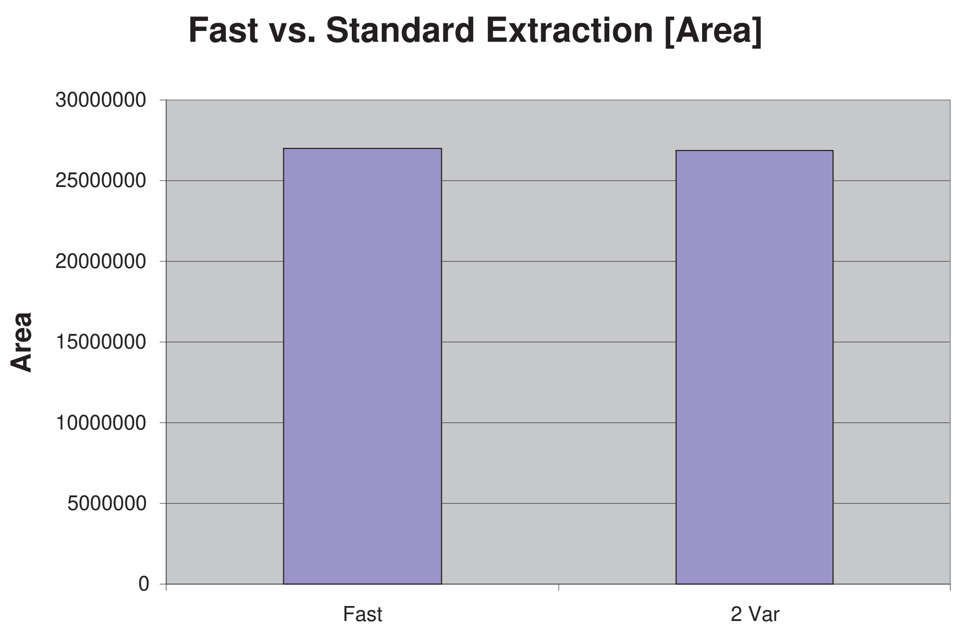
Figure 5.2: Area vs. Maximum Extractor Size.

5.4.2 Fast Two Variable Extraction

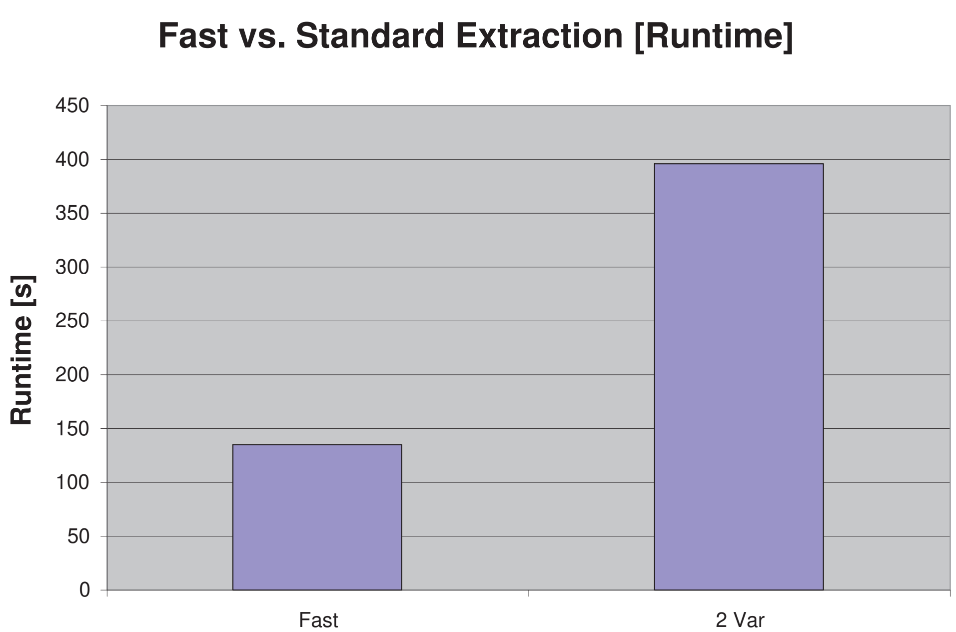
Further improvements in runtime are possible when the extractor size is fixed at two. Extractors can be found incrementally and transitively which alleviate the need to process extractors between all pairs of variables. While algorithmically faster, the fast extraction algorithm is inexact and may miss some sharing opportunities, however the loss was found to be minimal. In total, the fast extraction algorithm runs 2.5 times faster than the exact algorithm, while inflating area by merely 0.08%.

5.4.3 Sharing Extraction vs. No Sharing Extraction

Finally, to determine the impact that sharing extraction has on area, we obtain area results produced using FBDD both with and without sharing extraction enabled. For the best area and runtime balance, we use the fast, two variable sharing extraction algorithm



(a) Area



(b) Runtime

Figure 5.3: Fast vs. Exact Two Variable Extraction.

in these tests. The area results are shown in Figure 5.4. The benefit experienced from sharing extraction is highly dependent on the circuit type. Circuits *k2* and *vda* experiencing savings of over 100%, while a few circuits do not benefit from sharing extraction at all. Overall however, most circuits do experience benefit from sharing extraction, with the average area savings found to be a substantial 28%.

As an added benefit, our sharing extraction algorithm also improves the overall runtime of logic synthesis. The runtime results for FBDD with and without sharing extraction is shown in Figure 5.5. Adding sharing extraction capabilities to logic synthesis has resulted in a runtime improvement of 82%! This is possible because sharing extraction is interleaved with decomposition, which work together in breaking the circuit down into basic gates. Each transformation that is handled with sharing extraction means that one less decomposition is required. Our findings indicate that the computational cost of performing sharing extraction is less than the cost of decomposition. The result, is a synthesis system with both substantially improved area and runtime.

5.5 Scalability

Logic transformations are performed in a folded fashion where logically equivalent functions share logic transformations. To determine the effectiveness of this approach, we count the number of regular transformations, which are the number of transformations required in a non folded environment, and compare it to the number of folded transformations required.

5.5.1 Synthetic Benchmark

In this experiment we investigate the scalability of folded synthesis. We wish to see how folded synthesis performs as regularity is increased in a controlled manner. To do this we generate benchmark circuits by instantiating varying number of copies of a template

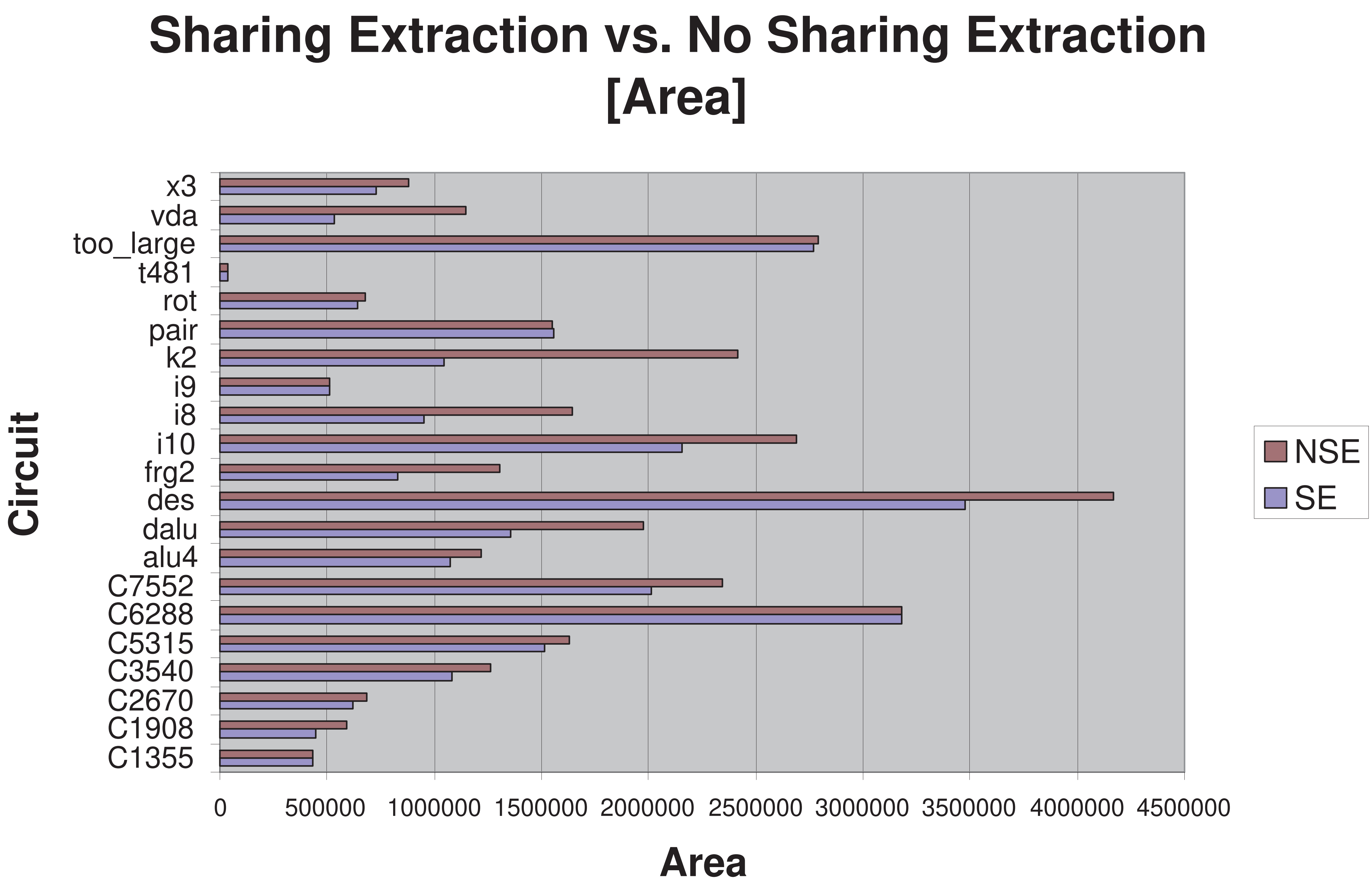


Figure 5.4: Sharing Extraction vs. No Sharing Extraction [Area].

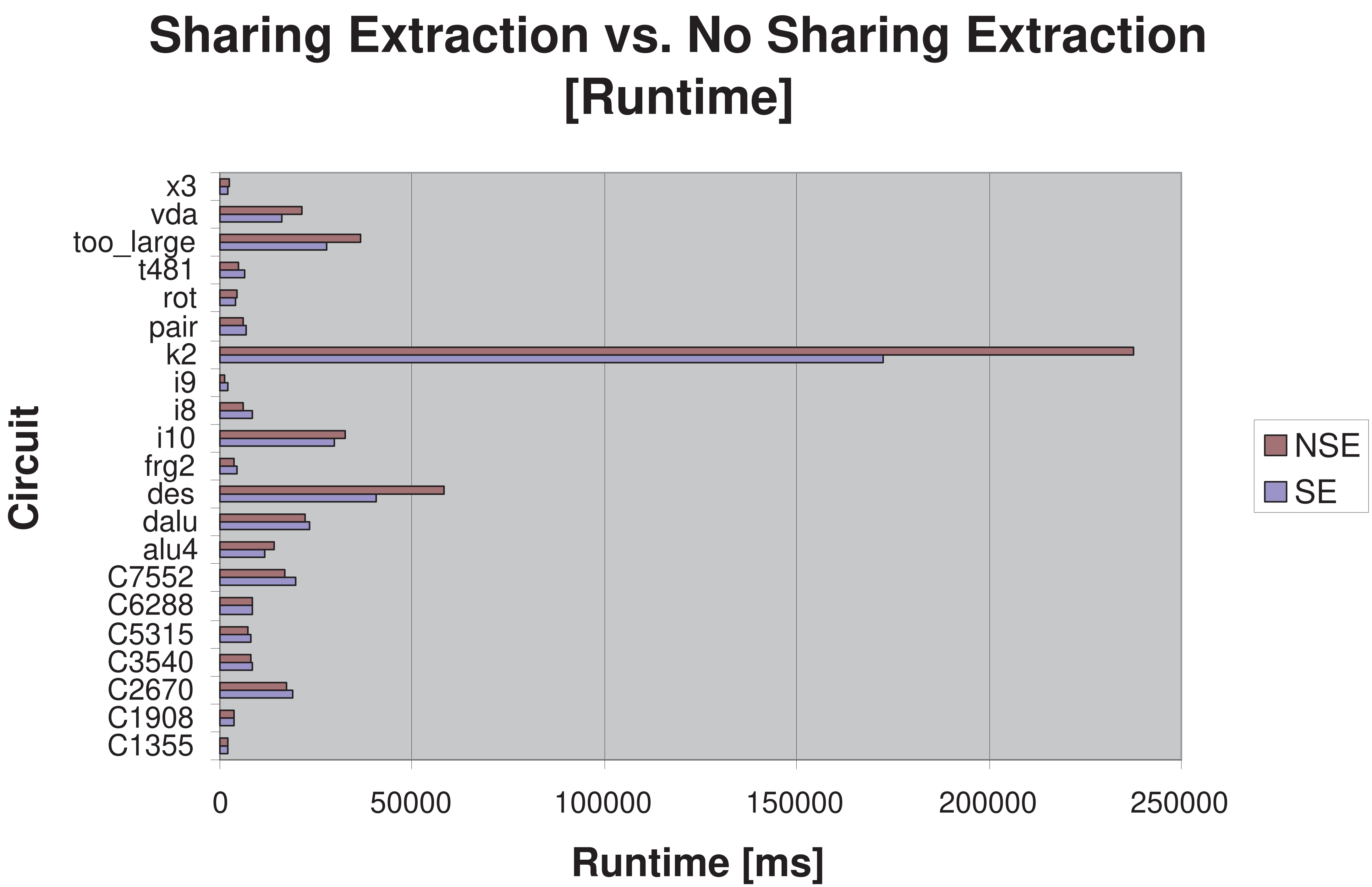


Figure 5.5: Sharing Extraction vs. No Sharing Extraction [Runtime].

circuit. For the template circuit we use rot.blif from the MCNC benchmark. In this way, regularity is increased while the logic content of the circuit remains the same. In total, there are ten benchmark circuits with the number of instances of rot varying from one to ten. We report the runtime growth for the elimination, decomposition and sharing extraction stages against this synthetic benchmark.

Elimination

The runtime growth of elimination is shown in Figure 5.6. The "Time" and "# of Folded Elims" plots show normalized values, which emphasize growth instead of absolute value, to enable their comparison. The normalized values are computed as $V(N)_{normalized} = V(N)/V(1)$, where N is the number of instances. A "Reference" line reflects the total time required for elimination if each instance of rot were processed individually. From the graph, it shows that "# of Folded Elims." remains constant for all circuits repetitions, due to the fact that additional instances can share the eliminations computed for the first.

The total runtime of elimination can be broken down into sharable and non-sharable components. In elimination, the sharable components consist of collapsing BDDs with the compose operation, and the simplification of the composed function that follows. The shared components are computed once, and the result shared with all compatible elimination pairs. The non-shared parts consist of computing the characteristic functions for the support configurations, and updating the gate instances in the Boolean network as eliminations are committed. As regularity is increased with each added repetition, the cost of computing the sharable components remain unchanged while the cost of the non-sharable components grows linearly. In practice, the sharable costs may grow as well because the sequence in which transformations take place in each template instance cannot be guaranteed to be the same. But in practice, any increase in the sharable costs are minimal.

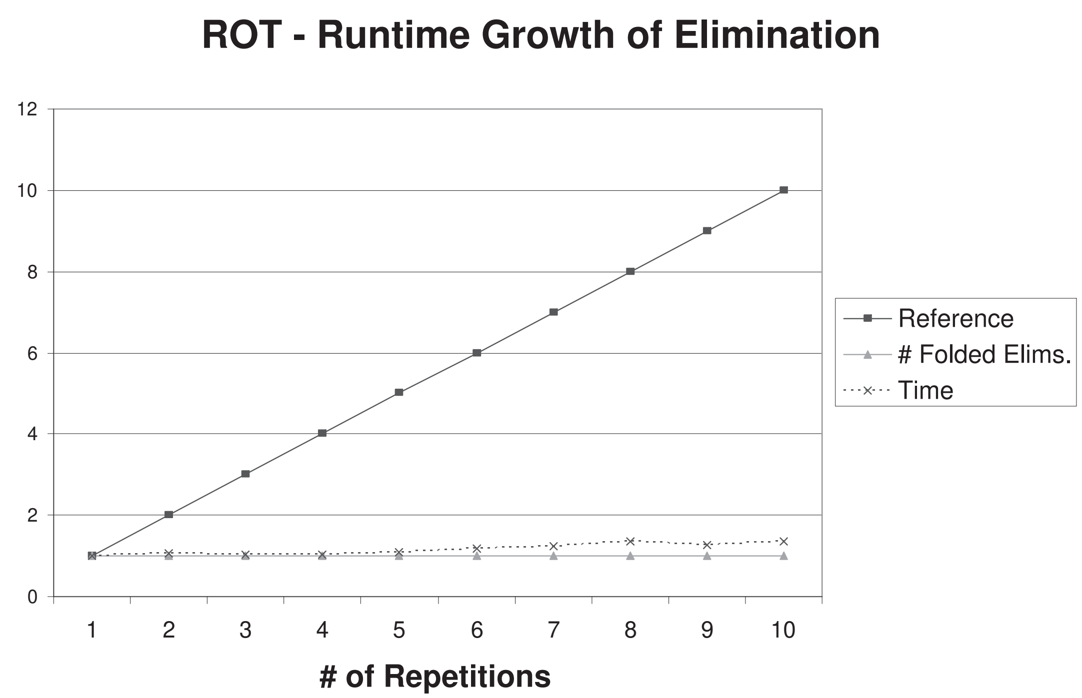


Figure 5.6: ROT.blif - Elimination.

The normalized values for the actual time, plotted in Figure 5.6, closely follows the "# of Folded Elims.". At 10 repetitions, the actual time has grown to a mere 1.36, illustrating that the sharable costs dominate the overall cost of elimination. The cost of the non-sharable component, while not negligible, grows far more slowly than if regularity were not used.

Decomposition

The runtime growth for decomposition, as shown in Figure 5.7 has characteristics similar to the growth for elimination. Again, the count for the number of folded transformations remains relatively constant for all numbers of repetitions. Although this time, the plot is not perfectly constant, due to differences in the way each instance is synthesized. For decomposition, the sharable portion consists of computing the various BDD decomposition algorithms. The non-sharable portion consists of updating the gates for each instance as

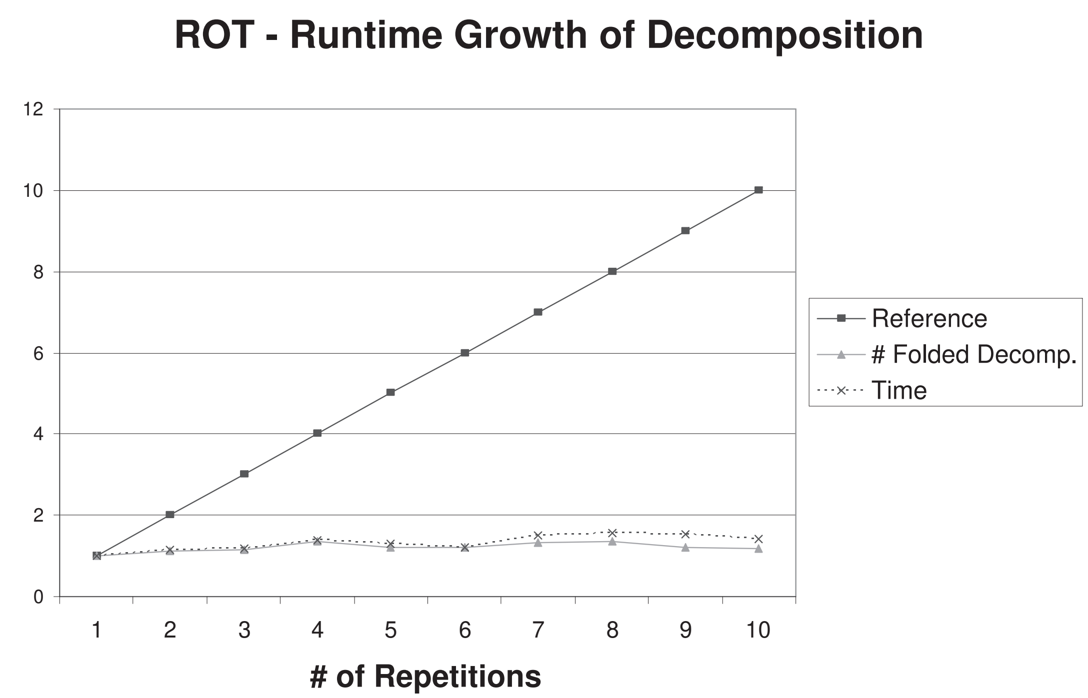


Figure 5.7: ROT.blif - Decomposition.

decompositions are applied. The non-sharable costs makes up an even smaller fraction of the total cost when compared to elimination where support configurations were computed. As a result, the growth rate for actual time spent on decomposition grows even slower than that of elimination. At 10 repetitions, only an 18% increase in the runtime of decomposition is experienced.

Sharing Extraction

Sharing extraction has two separate, sharable computations. The first sharable computation, called SE1 for reference, is the enumeration of disjunctive extractors. Equivalent functions will produce the same list of disjunctive extractors which can be shared by all instances of the function. This is the cost of computing cofactors between all adjacent variables of the function to determine if they can be disjunctively extracted. It does not include, however, enumerating the extractors in terms of absolute support, which must be performed for each gate individually.

The second sharable component, called SE2 for reference, is the computation of remainders. When an extractor is selected for sharing, it must be extracted from its parent function to produce a remainder, which requires the expensive process of simplification through variable reordering. For an extraction that breaks F down into remainder R and extractor E, extractions that involve the same F and use the same relative positions of the variables of E in F, produce the same remainders.

The growth for the number of folded computations for each of SE1 and SE2 are shown in Figure 5.8. The number of folded computations remains virtually flat for both plots. At 10 repetitions, only 10% more folded SE1 computations and 11% more folded SE2 computations are required. Due to the high non-sharable cost of manipulating large lists of extractors, the actual runtime grows quite noticeably. The run time of the sharing extraction component is doubled when synthesizing 10 instances. However, the overall growth is still far smaller than if each computation were performed individually.

Runtime Growth

The total, overall runtime, is shown in Figure 5.9. It has growth similar with the three major synthesis steps described earlier. 2403 ms were required to synthesize a circuit with 10 repetitions of rot.blif where 12800 ms would have been required if each instance were synthesized individually; a runtime savings of 81%.

5.5.2 MCNC

Here we look at how folded synthesis performs under a set of comprehensive benchmarks. We run the benchmarks through FBDD and count the number of folded and regular transformations performed. Dividing the number of regular transformations by the number of folded transformations gives an indication of the runtime improvement that can be expected. It is essentially an upperbound on the achievable speedup.

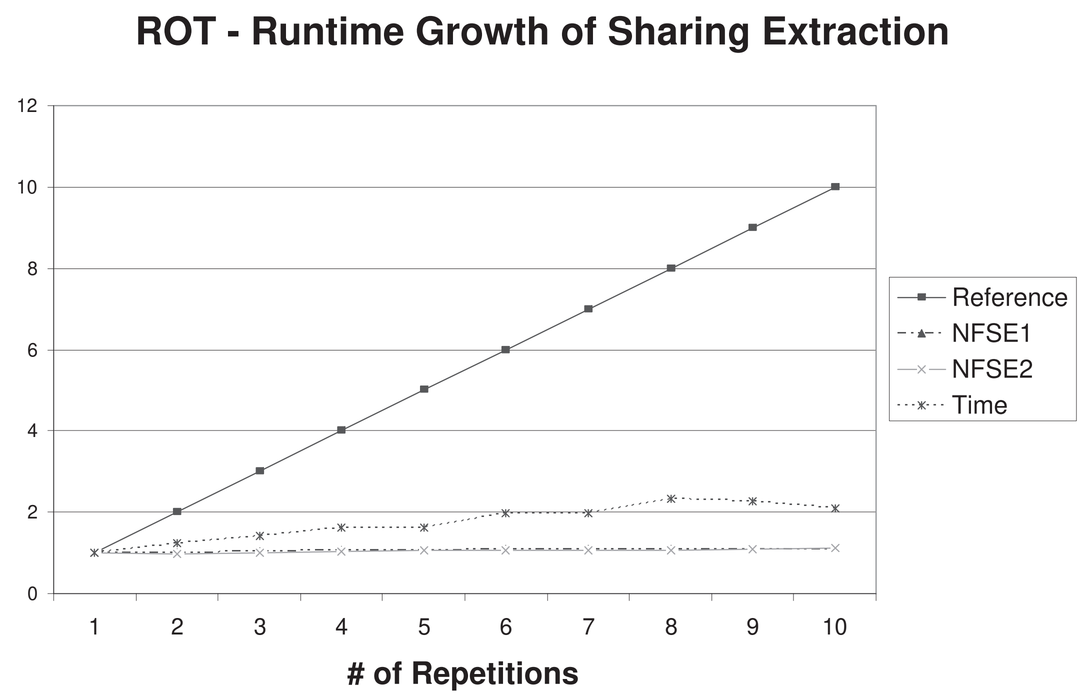


Figure 5.8: ROT.blif - Sharing Extraction.

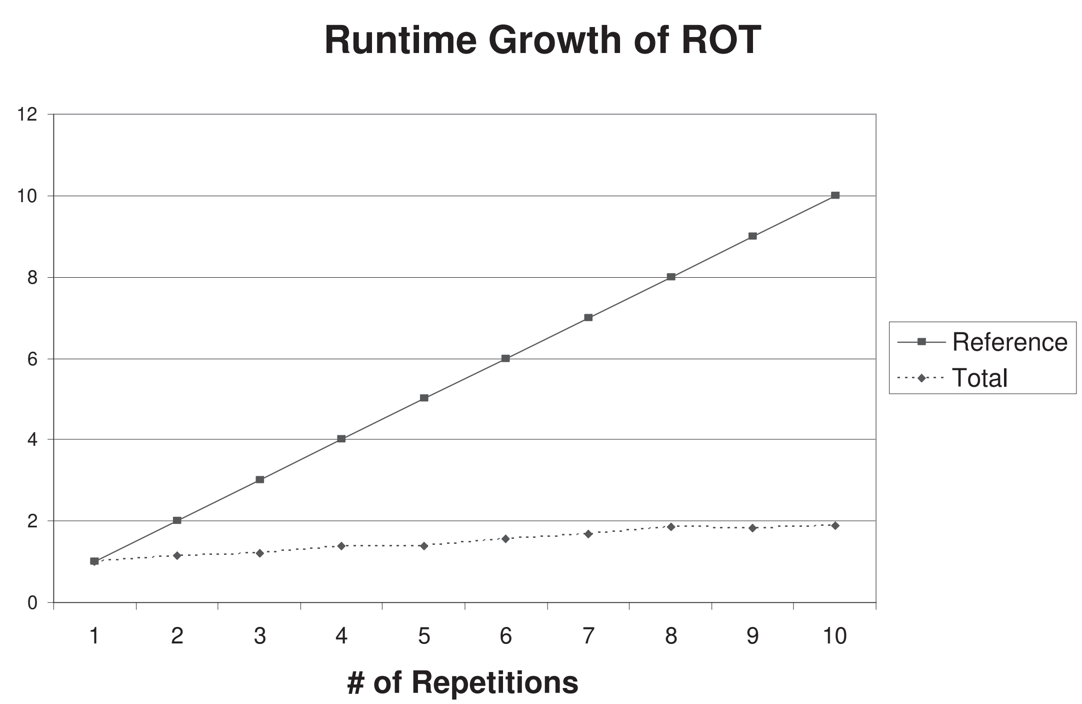


Figure 5.9: Runtime Growth of ROT.

Elimination

Figure 5.10 shows the number of regular versus folded eliminations counted. On average, the folded approach requires 3.85 times less eliminations than the regular approach. Reductions varied between 1.07 times in *alu4* to 136.25 times in circuit *C6288*.

Decomposition

Figure 5.11 shows the number of regular versus folded decompositions counted. On average, the folded approach requires 4.35 times less decompositions than the regular approach. Reductions varied between 1.73 times in circuit *alu4* to 30 times in circuit *C6288*.

Sharing Extraction

The number of regular and folded extractor enumerations are shown in Figure 5.12. On average, 11 extractor enumerations share one computation. The number of regular and folded remainder computations are shown in Figure 5.13. On average, 3 remainders share one computation.

5.6 Comparison with SIS and BDS

We run the benchmarks through FBDD and two other synthesis systems, SIS and BDS. SIS, a cube set based synthesis system, was developed at UC Berkeley. It has a flexible command line interface that allows users to try different combinations of optimization tasks. We run SIS using the well known *script.rugged* script. BDS, is a BDD based logic synthesis system, developed at the University of Massachusetts Amherst. We run BDS using its default options. To obtain area information, the optimized netlists are mapped to a standard cell library using the SIS mapper.

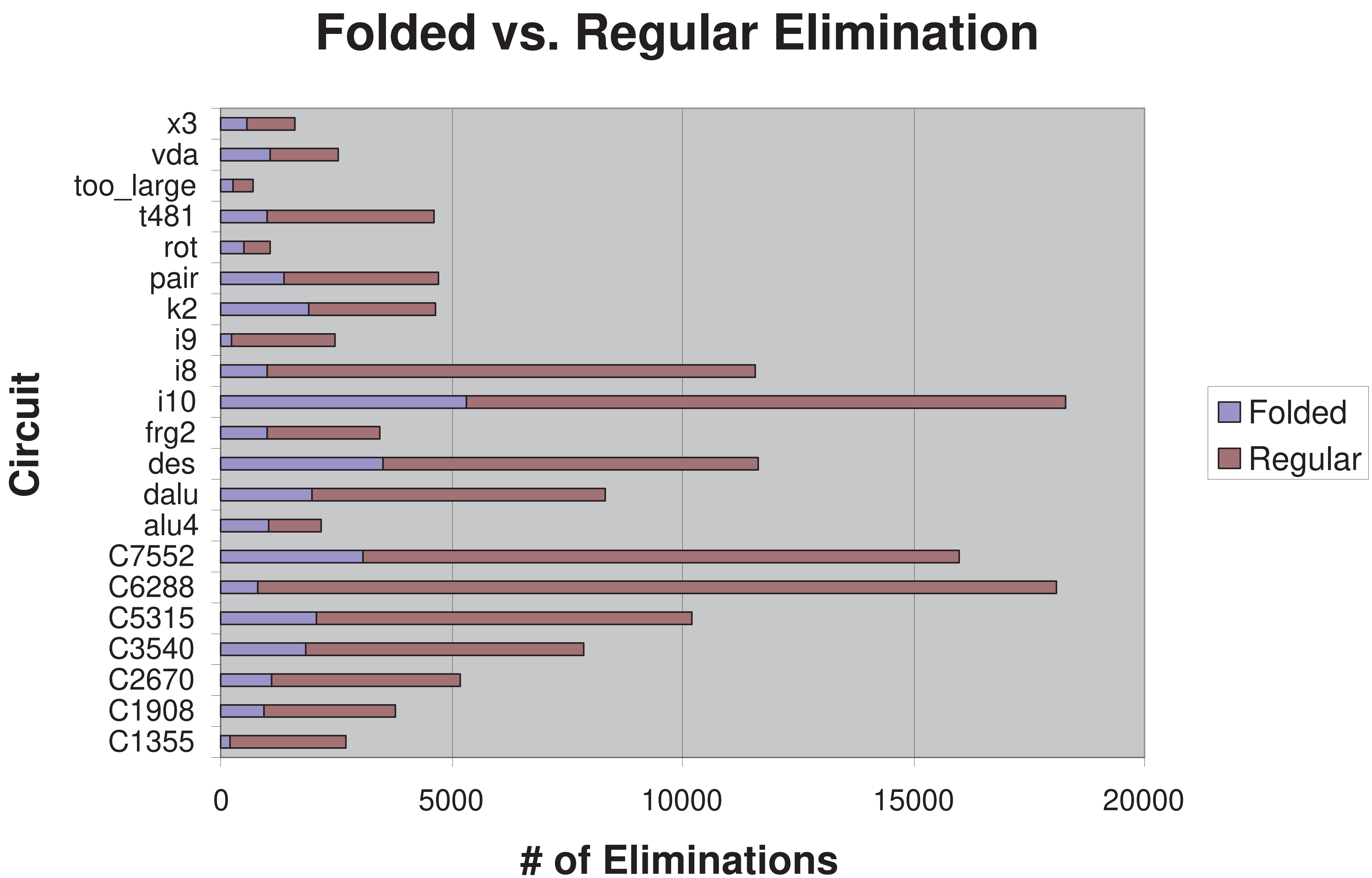


Figure 5.10: Folded vs. Regular Elimination.

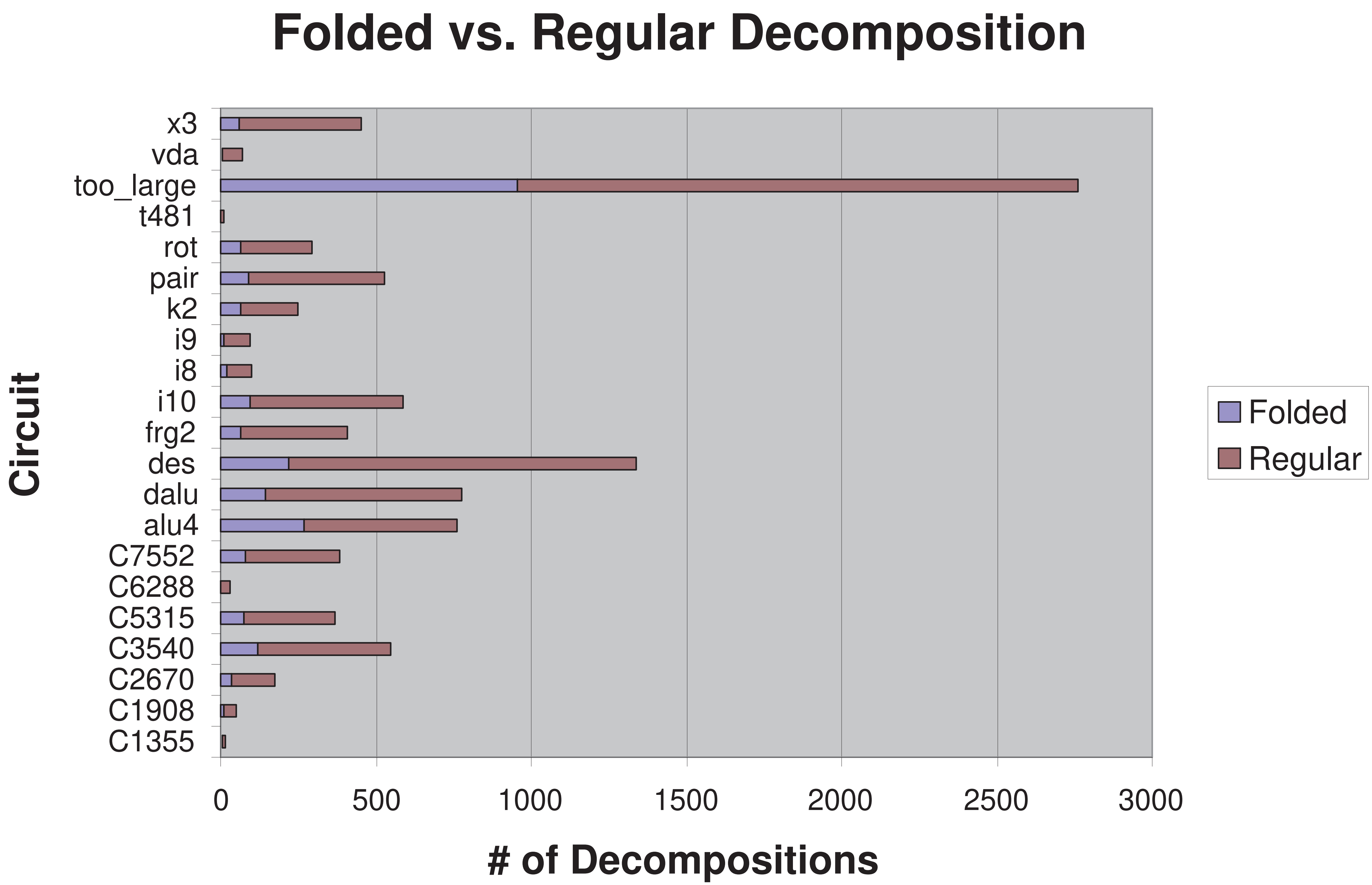


Figure 5.11: Folded vs. Regular Decomposition.

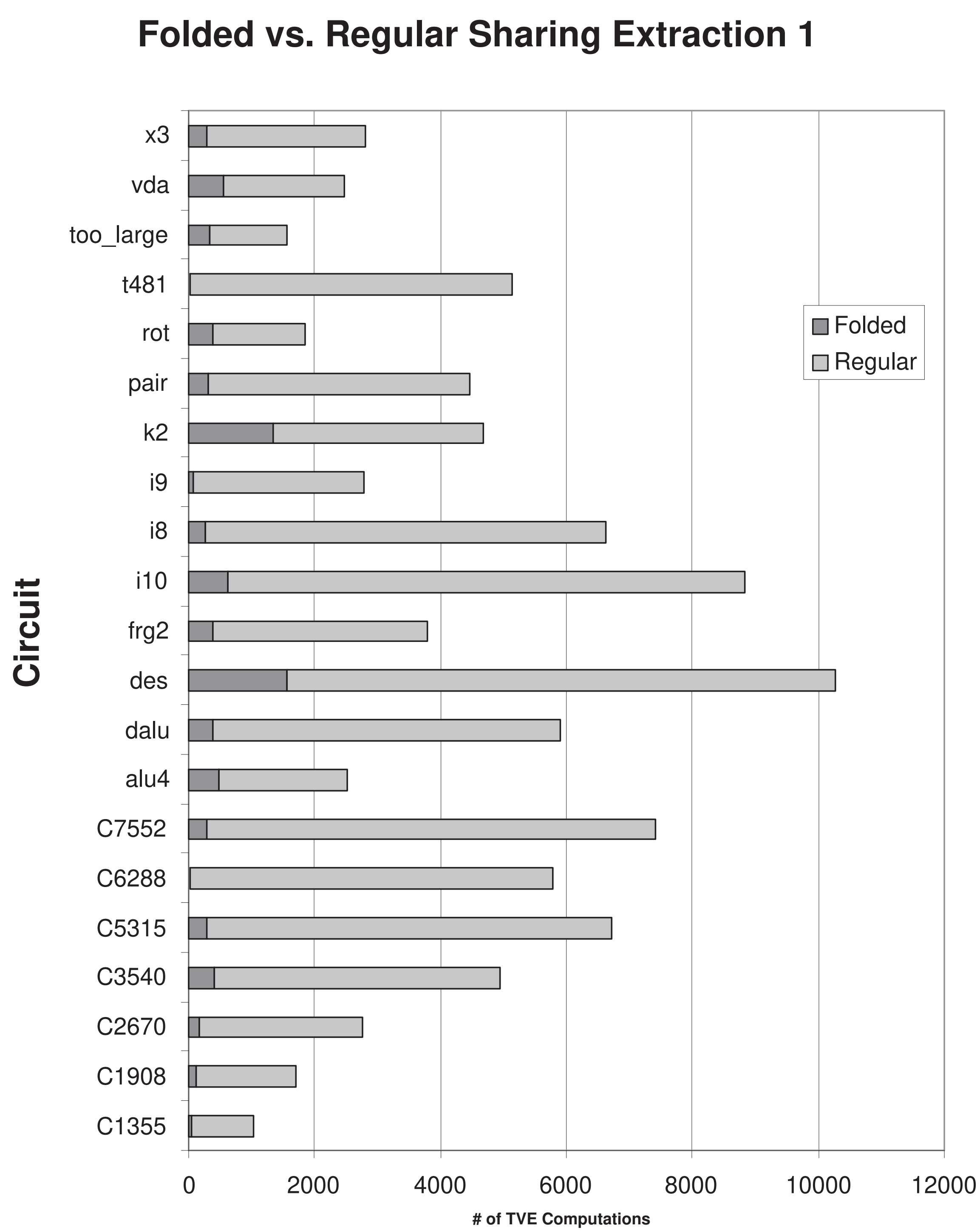


Figure 5.12: Folded vs. Regular Extractor Enumeration.

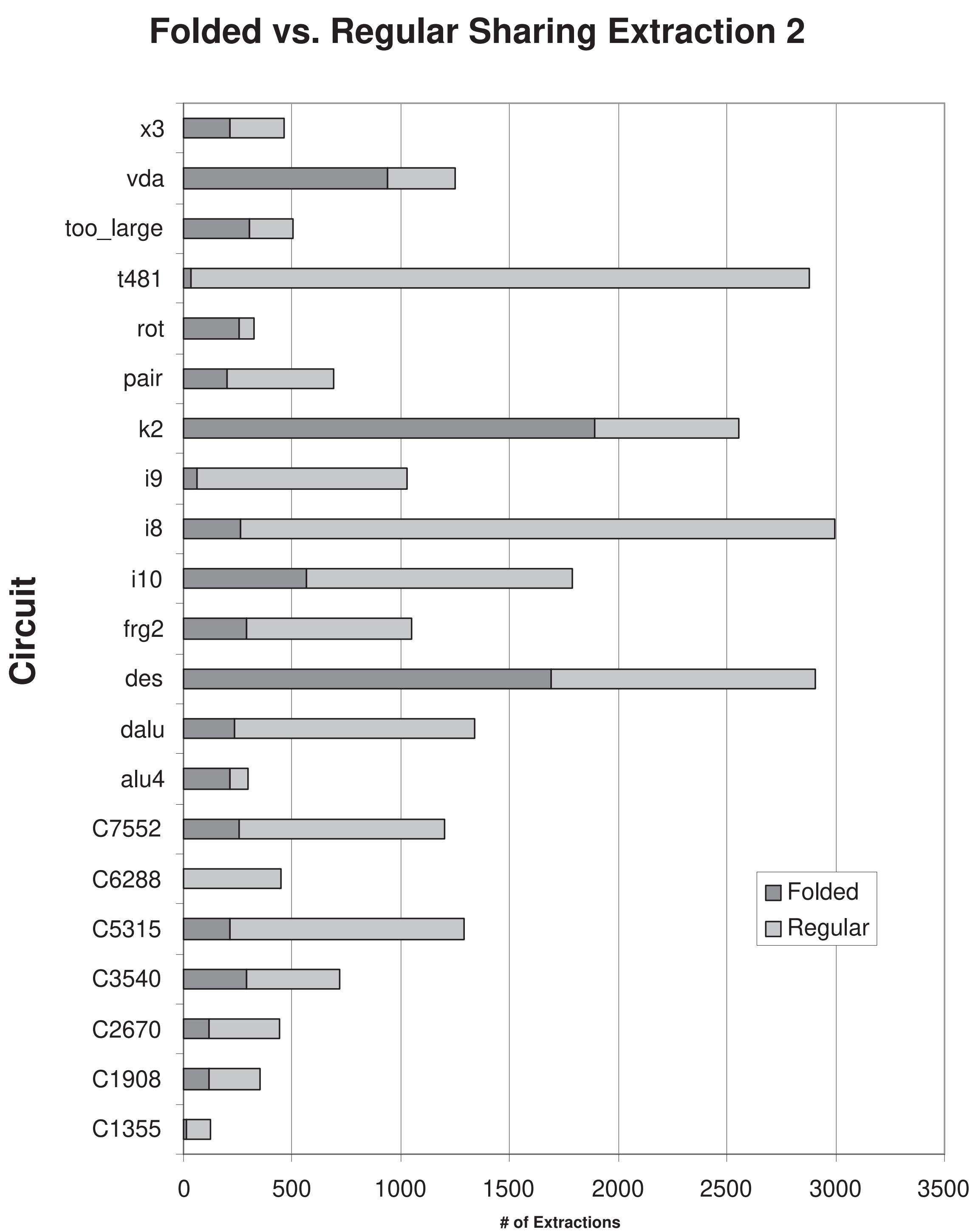


Figure 5.13: Folded vs. Regular Remainder Computation.

5.6.1 MCNC Benchmarks

The sharing extraction algorithm and regularity aware folded synthesis approach has enabled FBDD to have both dramatically improved area and runtime results over a previously state-of-the-art BDD based logic synthesis system BDS. The area results are shown in Figure 5.14. For test case, k2.blif, for which BDS was unable to run successfully, the area result produced by SIS was used in its place, for the purpose of computing averages. FBDD, on average, produces circuits with 21% less area than BDS. While a major improvement, FBDD still falls short of SIS which produced circuits with 15% less area.

In terms of runtime, FBDD clearly runs faster than BDS and SIS. The runtime results are shown in Table 5.15. For test case, k2.blif, which BDS could not synthesize, the runtime produced by FBDD is used in its place for the purpose of computing an average. Note that to accommodate SIS's runtime for too_large, the time axis was extended from 250,000ms to 2,500,000ms. On average, FBDD runs over 15 times faster than SIS and 3 times faster than BDS.

5.6.2 ITC Benchmarks

The ITC [6] benchmarks, developed at Politecnico di Torino, include subsets of the Viper and 80386 processor cores which offer test cases that are 13 times larger than those found in the MCNC benchmarks. We use the ITC benchmarks to evaluate the performance of the synthesis tools on large benchmarks. The results for FBDD and SIS are shown in Figure 5.16. BDS cannot not synthesize sequential circuits and could not be included in this study.

Due to the nature of processor cores, which contain a high degree of regularity FBDD was able to synthesize the ITC benchmark circuits in 12 times less time than SIS, on average. The average does not include circuits *b17* or *b17_1* which could not be synthesized

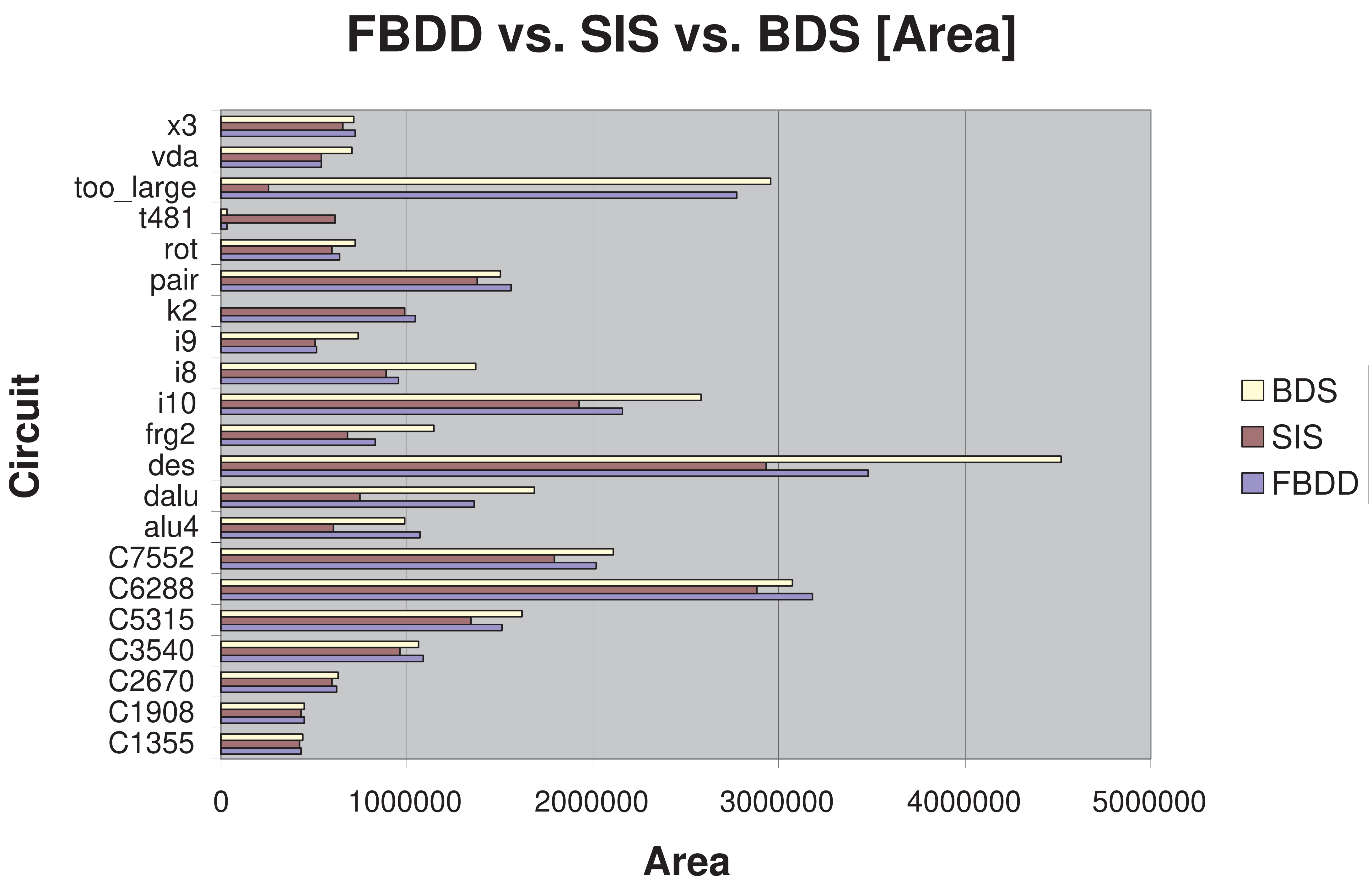


Figure 5.14: FBDD vs. SIS vs. BDS [Area].

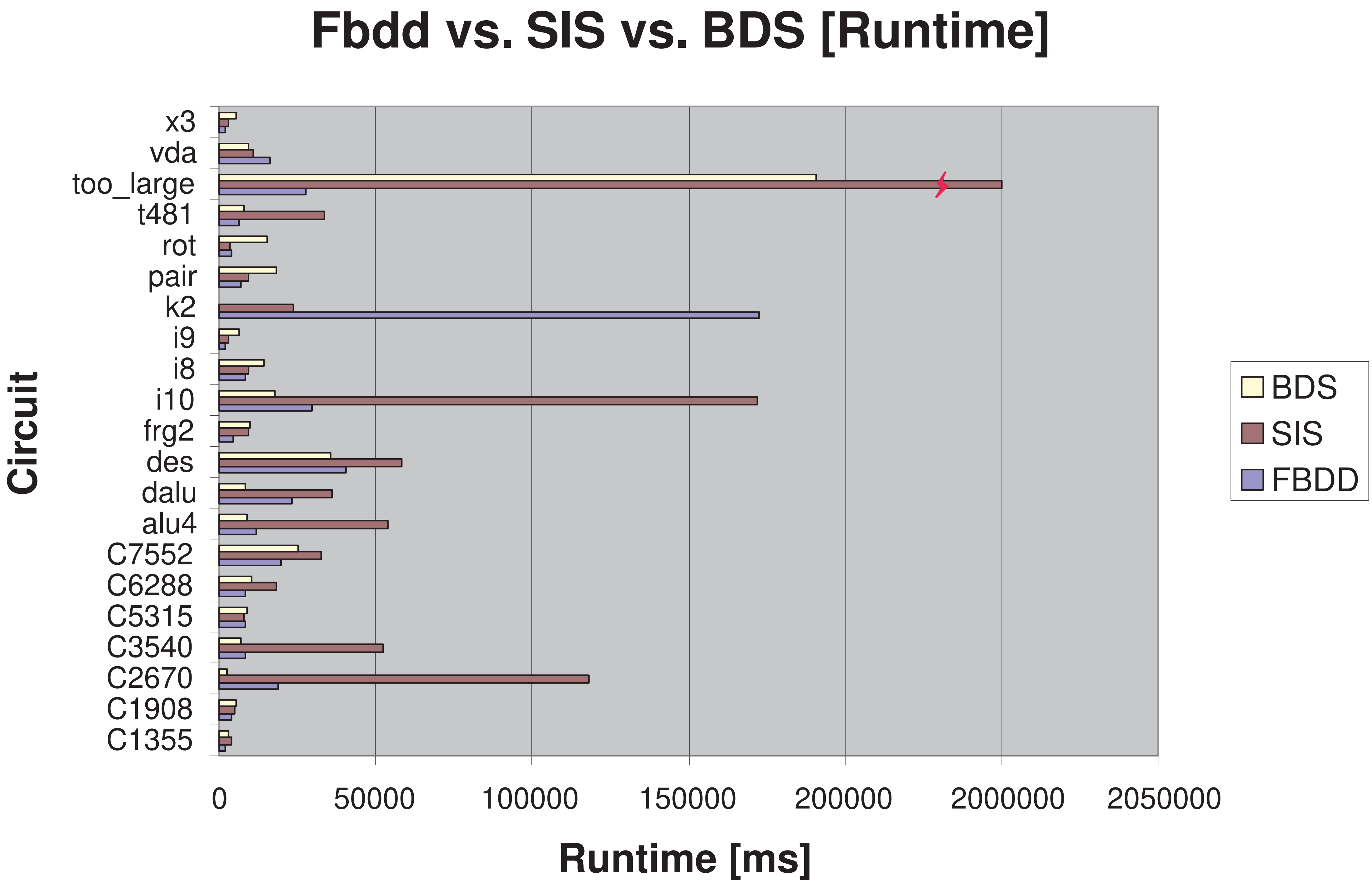
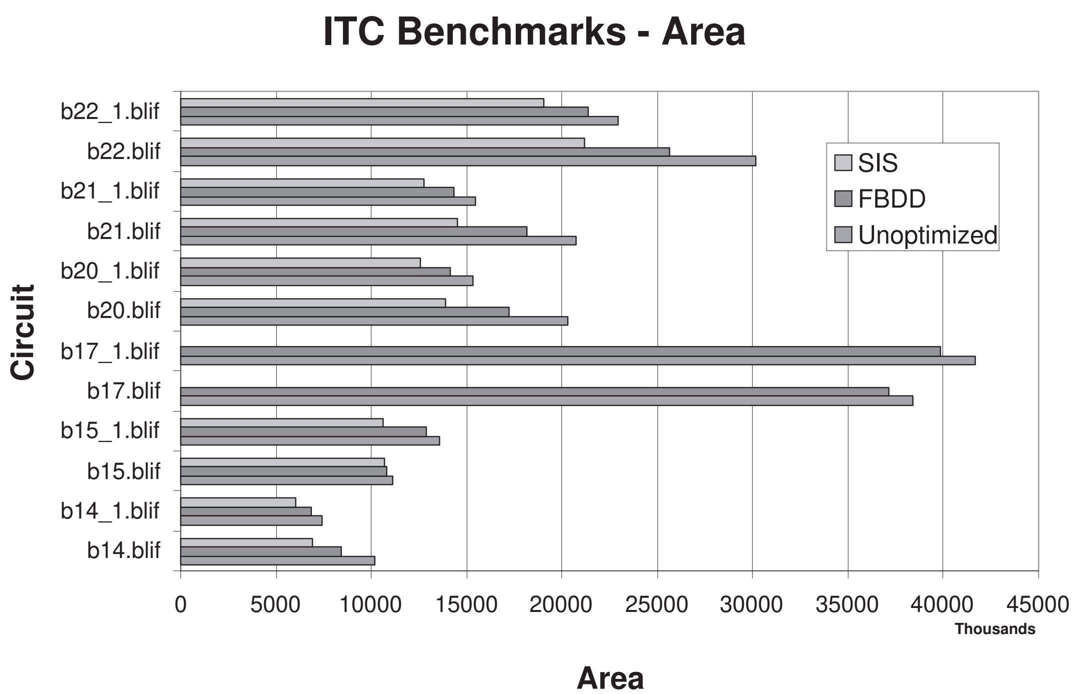


Figure 5.15: FBDD vs. SIS vs. BDS [Runtime].



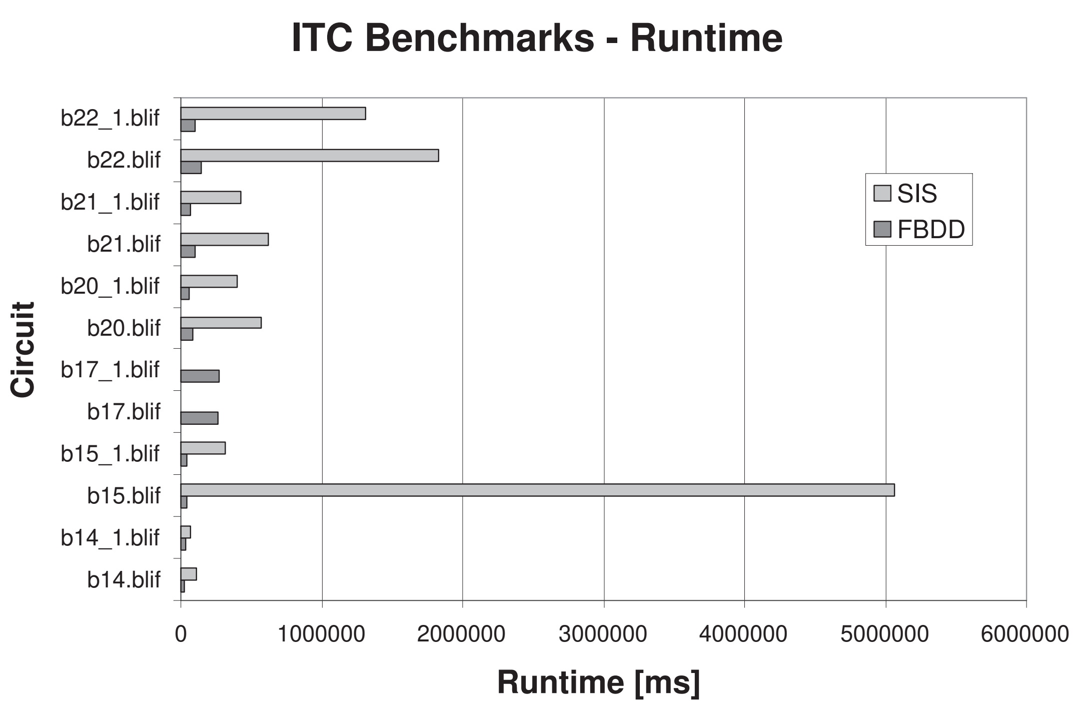


Figure 5.16: FBDD vs. SIS vs. BDS.

with SIS in under eight hours. As is consistent with the results found with the MCNC benchmarks, SIS produced better area by 16%.

5.6.3 Synthetic Benchmark

To evaluate the behavior of the synthesis tools as regularity is increased, we revisit the benchmarks used earlier where multiple instances of rot.blif are synthesized together. The result is shown in Figure 5.17. As shown earlier, regularity awareness enables FBDD to synthesize multiple copies of rot.blif together, much faster than would be possible if each instance were synthesized individually. FBDD requires a relatively small amount of runtime for each additional instance of rot.blif. On the other hand, BDS takes more time to synthesize the copies together than separately. Synthesizing one copy of rot.blif takes BDS 1420 ms, while synthesizing 10 copies requires 16950 ms. The growth of SIS's runtime is significantly worse requiring 75100ms to synthesize 10 copies together, when only 4300ms was required to synthesize one copy.

5.6.4 Adders

In this experiment, we compare the synthesis of ripple carry adders, to illustrate the importance of XOR gates in synthesis. The experiment contains four adders of varying bit widths. The adders begin completely collapsed and the tools are applied to demonstrate their ability to automatically extract XOR gates and find sharing to produce a solution of minimum area. The area results obtained are shown in Figure 5.18.

Both FBDD and BDS are able to find the area optimal ripple carry adder solution, given their ability to extract XOR gates. In this case, BDS's passive sharing extraction algorithm is sufficient in finding the sharing between output functions.

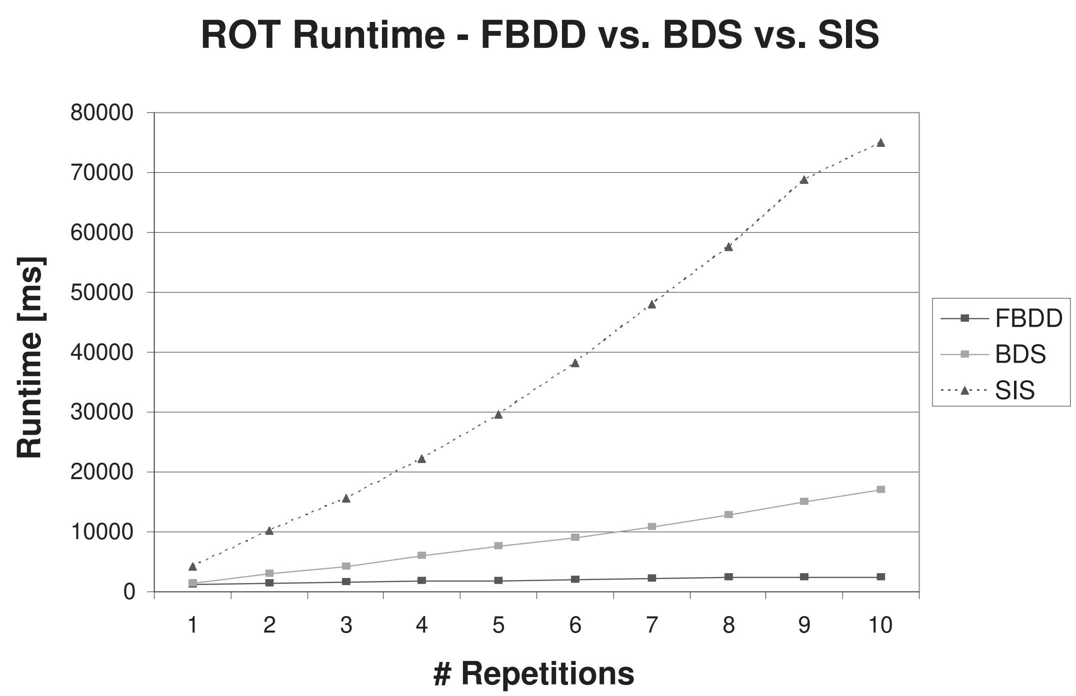


Figure 5.17: ROT.blif - Runtime comparison between FBDD,BDS and SIS.

SIS's solution leads to poor area quality. While SIS is able to find the sharing between the output functions, the solution it produces for the full adder units is an overly complex interconnection of AND and OR gates. Furthermore, the runtime characteristics is very poor, given the inefficient cube set representation for arithmetic circuits. The explosion in size of arithmetic type functions is a serious limitation for the cube set representation, and caused SIS to fail to complete the synthesis of a modest sized 16-bit adder in under eight hours. For this reason, arithmetic circuits are generally left untouched in cube set based synthesis systems, or are handled by special purpose generators. As the circuit types considered for automated synthesis continue to move from the traditional glue logic towards arithmetic and datapath intensive circuits, this difference will become more pronounced.

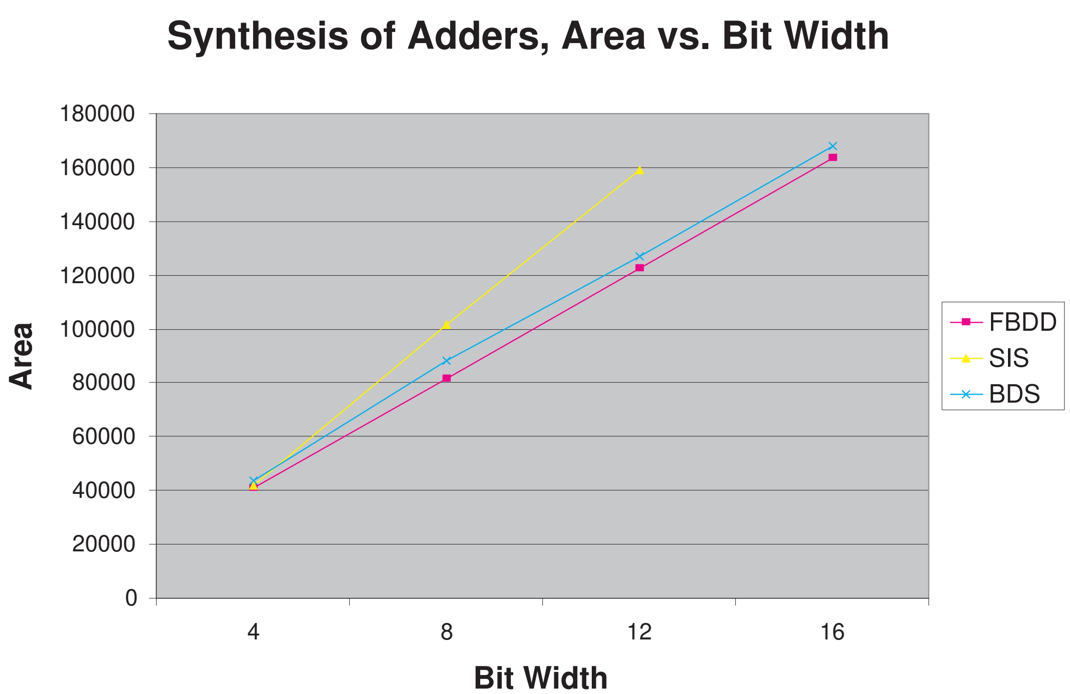


Figure 5.18: Synthesis of Adders, Area vs. Bit Width.

Chapter 6

Conclusion

In this thesis, a new sharing extraction algorithm and regularity based framework for BDD based logic synthesis was described. A key finding of this thesis is the use of two variable, disjunctive extractors as candidates for sharing extraction. With the BDD, these extractors can be found quickly by computing the result of a few cofactors. Old, disjunctive, two variable extractors remain valid in their remainder functions, enabling a fast, incremental solution. And disjunctive, two variable extractors are transitive, enabling us to reduce the search to extractors of adjacent variables, while sacrificing very little in terms of area quality.

The fast equivalence checking capability of the BDD has opened a new opportunity for regularity extraction. As regularity is in abundance, this regularity can be applied to share logic transformations to improve runtime.

While the area quality produced by FBDD continues to lag that of SIS, significant improvements over existing BDD based systems have been made. Fundamentally, BDDs are smaller and faster data structure to work with and better equipped to deal with arithmetic type circuits. As the algorithms of BDD based logic synthesis matures, they have the potential to significantly advance the state of the art and replace the cube set at the representation of choice.

Bibliography

[1] S. R. Arikati and R. Varadarajan. A signature based approach to regularity extraction. In *Proceedings of the International Conference on Computer Design*, San Jose, 1997.

[2] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *1997 IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[3] R. K. Brayton and C. McMullen. Decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, pages 49–54, 1982.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers, Vol. C-35, No. 8*, pages 677–691, 1986.

[5] J. Ciric and C. Sechen. Efficient canonical form for boolean matching of complex functions in large libraries. In *2001 IEEE/ACM International Conference on Computer Aided Design*, 2001.

[6] F. Corno, M. S. Reorda, and G. Squillero. RT-level ITC 99 benchmarks and first atpg results. In *IEEE Design & Test of Computers*, pages 44–53, 2000.

[7] 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.

[8] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design*, pages 126–129, 1990.

[9] D. Debnath and T. Sasao. Fast boolean matching under permutation using representative. In *Asia and South Pacific Design Automation Conference, ASP-DAC'99,2001 IEEE/ACM*, pages 359–362, 1999.

[10] S. Ercolani and G. D. Micheli. Technology mapping for electrically programmable gate arrays. In *28th ACM/IEEE Design Automation Conference*, 1991.

[11] K. Karplus. Using if-then-else dags for multi-level logic minimization. In *http://www.cse.ucsc.edu/ karplus/research.html*, 1988.

[12] T. Kutzschebauch and L. Stok. Regularity driven logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 439–446, San Jose, 2000.

[13] 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.

[14] J. Mohnke, P. Molitor, and S. Malik. Application of BDDs in boolean matching techniques for formal logic combinational verification. In *International Journal on Software Tools for Technology Transfer*, pages 48–53, 2001.

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

[16] T. Sasao and M. Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic and Synthesis*, pages 471–477, 1998.

[17] 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.

[18] H. Sawada, S. Yamashita, and A. Nagoya. An efficient method for generating kernels on implicit cube set representations. In *International Workshop on Logic and Synthesis*, pages 260–263, 1999.

[19] 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.

[20] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of bdd's using don't cares. In *Proc. Design Automation Conf.*, pages 225–231, 1994.

[21] F. Somenzi. CUDD: Cu decision diagram package release 2.3.1. Technical report, Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2001.

[22] 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.

[23] 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.