Contents

Spectral Techniques for Technology Mapping

Jerry Chih-Yuan Yang Giovanni De Micheli

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

Technology mapping is the crucial step in logic synthesis where technology dependent optimizations take place. The matching phase of a technology mapping algorithm is generally considered the most computationally intensive task, because it is called on repeatedly. In this work, we investigate applications of *spectral* techniques in doing matching. In particular, we present an algorithm that will detect *NPN-equivalent* Boolean functions. We show that while generating the spectra for Boolean functions may be expensive, this simple algorithm offers significant pruning of the search space. The algorithm is implemented as part of the *Specter* technology mapper, and results are compared to other Boolean matching techniques.

1 Introduction

The field of logic synthesis has been a topic of intense research in the past decade. In particular, the technology dependent optimizations, or *technology mapping*, have been especially of interest. The existing work can be categorized into two basic approaches: The rule-based approach, as done in Socrates [4] and the algorithmic approach. There are two classes of algorithmic approaches: the pattern-matching approach, based on the works of Keutzer [6] and Detjens[2]; and the Boolean approach, based on the work of Mailhot [7]. The Boolean algorithmic approach is of theoretical and practical interest because of two main reasons: first, it enables optimizations of networks with a reduced dependency on the network structure or topology; second, it can explore more of the optimization space by including the *don't care* conditions that are defined either intrinsically or externally in the network.

One of the critical steps in a technology mapping algorithm is the *matching* phase. In this phase, a candidate network (or subnetwork) is matched against a set of library elements. In this work, we consider Boolean matching, where two functions are compared for functional equivalence. In [7], two functions match if they are *NPN-equivalent*.

In this work, we use the same definition of matching. We explore application of spectral techniques in the matching phase of a technology mapping algorithm. *Spectral techniques* have been used extensively in the design of threshold logic [1] and in the area of testability [5]. It is well known that spectral techniques offer a powerful way to classify Boolean functions[3]. However, because of their computationally intensive nature, spectral techniques have not received as much attention in recent years.

The matching phase of technology mapping is considered to be the most computationally intensive part of the process. The problem of matching functions using Boolean techniques has a worst-case complexity of $O(n!2^n)$ because of the need to try all $n!$ permutations of the input variables and 2^n phase assignments. Therefore, efficient pruning of the search space is necessary. In this work, we utilize the classification powers of the spectrum to detect Boolean equivalence between two functions. Previously, *binary decision diagrams (BDDs)* and *symmetry sets* were used to effectively reduce the complexity [7]. Spectral techniques also have been utilized to perform matching in [8].

We present an algorithm of detecting two *NPN-equivalent* Boolean functions based on comparing their spectra. This method is shown to be effective for libraries with elements of limited number of inputs (inputs less than 10). The rest of this paper is organized as follows. Section 2 gives background definitions for spectra and their operations. In Section 3, we define XNPN-equivalent class operations. Section 4 presents a matching algorithm for NPN functions. Results and concluding remarks are contained in Section 5.

2 Background

Various orthogonal transformations can be used to transform one Boolean function into an unique representation in the spectral domain. Many of such transforms are surveyed by Hurst et. al. [5]. In particular, the Hadamard transform is susceptable for computing purposes because a fast transform exists (much like the butterfly algorithm for the Fourier Transform) [5].

2.1 Hadamard Transform Properties

The Hadamard transform is an orthogonal matrix with the following recursive definition.

$$T^0 \equiv 1$$

$$T^n \equiv \begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix}$$

Let z be the output truth table of an n variable Boolean function F. To obtain the desired spectral domain representation, an initial recoding of z to vector y is performed. We first apply a recoding transformation $t : B \rightarrow B^t$ to every entry in the truth table vector z, where $B = \{0, 1\}, B^t = \{1, -1\}$. Then, each element in vector y is $y_i = t(z_i) = 1 - 2z_i$.

The spectrum s, a vector with 2^n elements, is calculated by

$$T^n \cdot y = s$$

Vector s has the property that $\sum_{i=0}^{2^n-1} s_i = \pm 2^n$.

As an example, a Boolean function of 3 variables ($n = 3$) generates a Hadamard spectrum with the following coefficients. The indices indicate the *order* of the coefficients.

$$s = \begin{bmatrix} s_0 & s_1 & s_2 & s_{12} & s_3 & s_{13} & s_{23} & s_{123} \end{bmatrix}^T$$

Each coefficient in the spectrum gives some global information about the Boolean function. The indices of a coefficient represent which input variables the coefficient correlates to. In the general case, for a given F, there is one *zeroth order* coefficient s_0. This term reflects the correlation of F to a constant value. This term is computed by $num\_false\_minterms - num\_true\_minterms$, where false minterms are 0 truth table entries, and true minterms are 1 truth table entries. Thus, if the function is a constant one, then $s_0 = -2^n$. There are n *primary first order* coefficients, $s_1, \ldots, s_n$. These coefficients show the likeness between F and input variable x_i. The orders of coefficients increase up to *nth order*. For higher order coefficients, the coefficient correlates to the *exclusive or* of all the input variables specified in the index. For example, s_{ijk} is a third order coefficient, showing the correlation between F and $x_i \bigoplus x_j \bigoplus x_k$. For any given coefficient and its indices, the coefficient is numerically equal to

($\sum$ agreements between F and function formed by the exclusive-or of the indices) – ($\sum$ disagreements between F and function formed by the exclusive-or of the indices)

The number of coefficients in the *kth order* is $\left(\begin{array}{c} n \\ k \end{array} \right)$. For a function of n inputs, there are $\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) = 2^n$ coefficients.

It can be verified that T^n is orthogonal and Hermetian. Therefore, its inverse can be easily computed. The *inverse* of T^n is found by

$$[T^n]^{-1} = 1/2^n \cdot [T^n]$$

Given a spectrum s, the coded Boolean domain truth-vector y can be recovered by $1/2^n T^n \cdot s = y$. Vector z can be obtained directly from y by applying the inverse of the recoding procedure done initially. Manipulations between spectral and Boolean domains are easy since forward and inverse operations involve multiplication of the same transform. Computation of the Hadamard Transform is accomplished by a fast algorithm. The algorithm is illustrated in Figure 1[5]. The number of computations required is $O(n \cdot 2^n)$.

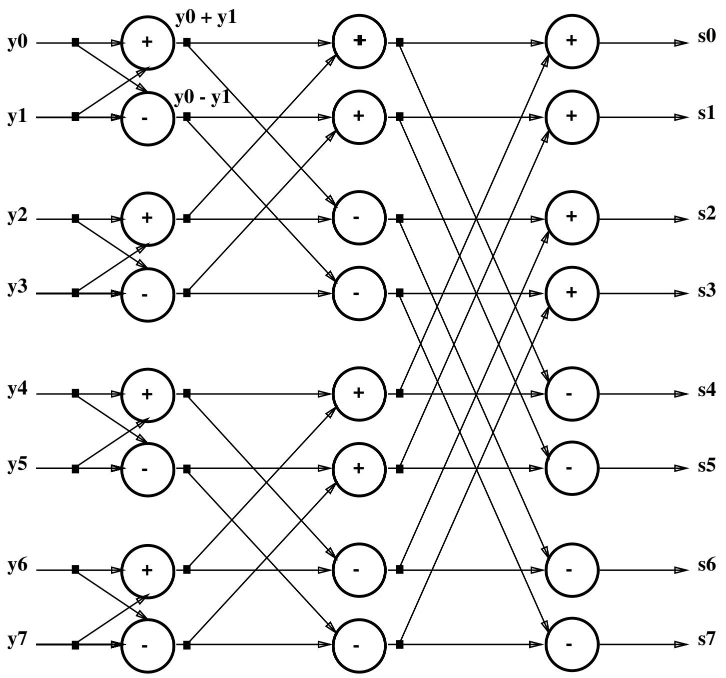


Figure 1: Hadamard Fast Transform Algorithm for 3-input Functions

2.2 XNPN Transformations

There are five transformations defined for the Hadamard transform to classify Boolean functions. Each of these transformations corresponds to an operation in the truth-table domain. We define the class of Boolean functions that are closed under these five transformations to be *XNPN equivalent.* In the paper by Edwards [3], the same classification is termed *disjointly translationally equivalent.* Proof of these operation being closed in a class of Boolean functions can be found in [1]. The transformations are:

1. Permutation of any two input x_i and x_j. In the spectrum domain, all coefficients containing i are swapped with those containing j;

$$s_i \text{ is swapped with } s_j$$
$$s_{ik} \text{ is swapped with } s_{jk}$$
$$\ldots$$

2. Negation of any input variable x_i. In the spectrum domain, any coefficient containing i is negated; namely,

$$s_i \text{ is replaced with } -s_i$$
$$s_{ik} \text{ is replaced with } -s_{ik}$$

$$\ldots$$

3. Negation of output variable. This involves negation of all 2^n coefficients

4. Replace any variable x_i into a network with $x_i \bigoplus x_j$.

$$s_i \text{ is swapped with } s_{ij}$$
$$s_{ik} \text{ is swapped with } s_{ijk}$$
$$\ldots$$

5. Replace output $f(X)$ with $f(X) \bigoplus x_i$.

$$s_i \text{ is swapped with } s_0$$
$$s_{ij} \text{ is swapped with } s_i$$
$$s_{ijk} \text{ is swapped with } s_{ik}$$
$$\ldots$$

The notion of a *NPN-equivalence* class can be defined as follows. An *NPN equivalent class* of Boolean functions is a set of functions that is closed under the first *three* transformations.

In other words, if two Boolean functions belong to the same NPN-equivalent class, then we can implement one function from the other by adding inverters to some inputs, and/or permuting some inputs, and/or negating the output.

In addition, when considering XNPN equivalence, one function can also be derived by another in the same class with additional *exclusive-or gates*. Clearly, functions belonging to a given NPN-equivalent class are a subset of the functions belonging to the same XNPN-equivalence class. Therefore, XNPN operations can classify a broader class of equivalence functions.

Using these operations, a given spectrum can be transformed into a *canonic* form. The canonic form of a spectrum is obtained by repeated application of the above operations to obtain an XNPN representation of the original spectrum with the following property. The first $n + 1$ coefficient values, from $s_0 \ldots s_n$, are arranged in decreasing magnitude order, where the magnitude of s_0 is the largest in the entire spectrum. These coefficients are made positive.

The canonic form can be used to uniquely identify an XNPN class. The conciseness of this classification technique is significant. For $n \leq 4$, all 65,536 Boolean functions are contained in 8 canonic classes [5].

3 NPN Matching Algorithm

Operations 4 and 5 in the previous section can be used to move coefficients across orders. In the truth-table domain, these operations correspond to adding exclusive-ors to the inputs or outputs.

We do not consider using these two operations in this work because of the complexity that the exlusive-or gates add to the resulting network. Therefore, a *Boolean match* in the context of this paper is defined as follows: Given a library function L and Boolean function F, a *match* exists if L and F are NPN-equivalent.

Two functions F and L are XNPN-equivalent, and therefore NPN-equivalent, if the corresponding spectra s_F and s_L are equal, modulo a permutation and complementation of their coefficients.

Note that none of the NPN operations can change the *order* of a coefficient. From the previous section, only the operations involving the exclusive-ors can translate a coefficient to a different order. Therefore, two functions are NPN-equivalent if the corresponding spectra are equal modulo complementation and permutation of the coefficients within the same order.

Another observation is that the sequence in which the NPN transformations are performed is irrelevant. The transformations are completely independent from one another. Performing one transformation will not affect the results of other transformations. In this matching algorithm, *negation* of the output is tried first. Then, *permutations* of inputs are performed, with the *negation* of inputs done as the last step.

During matching, only one spectrum is being modified. The idea is to manipulate the spectrum of the library element until it equals the spectrum being matched. The changes made to the library spectrum can then be added to implement the desired function. The algorithm does not distinguish between the two spectra. Figure 2 demonstrates the matching algorithm. The procedure for matching two spectra with n variables is as follows.

```
spectrum_match(s1, s2)
{
    c_graph = generate_NPN_compatibility_graph(s1, s2);
    if (s1_0 and s2_0 differ in sign) then
        negate each term in s1;
    generate_possible_input_orderings(c_graph);
    for each input ordering do begin
        for each input variable s1_i do
            if (s1_i and s2_i differ in sign) then
                negate all s1 terms with i in coefficient;
        if (s1_i equals s2_i for all i) then
            return (input ordering, phase assignment);
    end
    return (no match found);
}
```

Figure 2: Spectrum Matching Algorithm

3.1 Inversion of Output

No possible NPN operations can change the magnitude of the zeroth order coefficient. Therefore, if the magnitude of the zeroth-order coefficient of the two spectra is different, then the two functions cannot be NPN-equivalent, and matching is aborted. If the magnitude matches but the sign differs, then the sign of the zeroth order coefficient needs to be changed. To do so, all 2^n coefficients must be negated. This operation corresponds to the *negation* of the output.

A special case occurs when the zeroth order coefficient is 0. In this case, the functions are matched first without output inversion. If that fails, then matching is done with output inversion.

This extra overhead does not occur often since spectra with 0 as its zeroth order coefficient usually represent functions with an equal number of true minterms and false minterms (e.g. multiplexers and exclusive-or gates). With xors, a match will be found when comparing the input coefficients since negating an input is equal to negating the output. For multiplexers, similar input reorder and inversion will result in matches without having to negate output.

3.2 Permutation of Inputs

An *NPN compatibility graph* for first order coefficients is constructed. The graph is *bipartite*, defined to be $G(S, T, E)$, where S and T are sets of vertices; $S \cup T = V$, and $S \cap T = \emptyset$. $|S| = |T| = n$. Each set of vertices corresponds to a set of first order coefficients of a spectrum. Each vertex represents one coefficient. An edge $e \in E$ between $s \in S$ and $t \in T$ exists if the magnitudes of the corresponding coefficients are equal.

Each first order coefficient represents an input of the function. Thus, the edges correspond to possible locations where a given variable can be permuted. Note that this graph is concerned with possible permutations only; and phase assignment of the variables is dealt with later. Every vertex in S and T must have an edge incident upon it. This is an obvious requirement, since very coefficient must be matched. Therefore, the matching process aborts if there is an unmatched vertex in either S or T.

Matching the first order coefficients represents only a necessary condition. When a first order coefficient is modified, coefficients in the higher orders containing the corresponding index are modified as well. For example, suppose s_1 is swapped with s_2. Then, in subsequent higher orders, every occurrence of 1 in the indices of the coefficients is replaced by 2, and vice versa. This is illustrated by the permutation transformation in Section 2. Since we limit ourselves to the use of the first three transformations only, higher order coefficients are changed only by manipulation of the first order coefficients. The higher order coefficients must be checked for magnitude equality in order for a permutation to be valid.

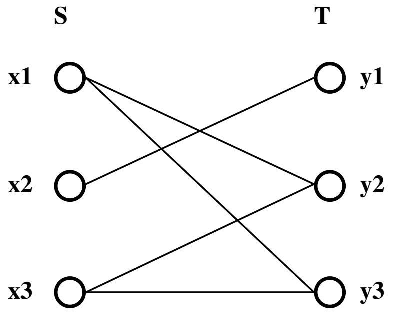
Let us establish the bound on the number of permutations for matching two n variable functions. Each permutation represents a possible ordering of the input variables. Let m be the number of

L = x1 x2' + x2 x3'

F = y1' y3' + y2' y3

	S_L	S_F
s0	0	0
s1	–4	0
s2	0	–4
s3	4	4
s12	4	–4
s13	0	–4
s23	4	0
s123	0	0

Compatibility Graph



Possible Permutations

	y1	y2	y3
1.	x2	x1	x3
2.	x2	x3	x1

Figure 3: Compatibility Graph for Matching 2 3-input Functions

distinct magnitudes in the first order coefficients. Let $S_i, 1 \leq i \leq m$ be the set that contains all coefficient with magnitude M_i. The number of permutations is

$$P = \prod_{i=1}^{m} |S_i|!$$

Note that m and all $|S_i|$ must be the same for both spectra, since an one-to-one match betwen S and T is required. Also note that $\sum_{i=1}^{m} |S_i| = n$. In the best case, if $m = n$, then each $|S_i| = 1$ and $P = 1$. This case occurs when all first order coefficients are unique; and therefore there is only one possible input ordering that can be tried. In the worst case, if $m = 1$, then each $|S_i| = n$ and $P = n!$. This case corresponds to all first order coefficients having the same magnitude; therefore each variable can possibly permute to all other positions. In practice, most spectra have $m \geq 2$. This reduces the number of permutations that have to be compared.

For example, in Figure 3, the spectra for s_L and s_F are shown. The corresponding compatibility graph for the first order coefficients is also shown, annotated by input variables. Assuming that set S is being permuted, the possible permutations are enumerated. Note that the number of permutations for this example is $P = 2! \cdot 1! = 2$. The first ordering of switching x_1 and x_2, along with inverting x_3, will result in a match of L with F.

When an input ordering is generated from the compatibility graph, it is checked for a possible match. The step consists of traversing the spectrum coefficients, performing the *permutation* invariant operation (Section 2.2). This operation requires swapping 2^{n-2} pairs of coefficients.

3.3 Phase Assignment of Inputs

While performing permutation for a given input ordering, we perform the phase assignment of the input variables at the same time. For a given input ordering, the coefficient (representing an input variable) is checked against the corresponding coefficient in the other spectrum. At this stage of the algorithm, the magnitudes of these two coefficients are the same as guaranteed by the compatibility graph. If the signs of coefficients are different, then the spectrum being modified performs the *negation of input* invariant operation on the input in question. This operation requires changing the sign of 2^{n-1} coefficients.

If there are no first order coefficient with the value zero, then a linear traversal of each input can guarantee correct phase assignment. Thus, a worst case n input negation operations need to be performed.

A special case occurs when there are first order coefficients equal to 0. In this case, a linear traversal will not guarantee correct phase assignment since the coefficient can take on both on and off phases. The number of input negation operations that need to be performed now becomes $(n-m)2^m$, where m is the number of 0 first-order coefficients.

Lastly, the modified spectrum is compared term-wise against the spectrum to be matched. If the spectra are equal in both magnitude and sign, then a match is found. This step entails 2^n comparisons, where n is the number of inputs.

3.4 Computational Complexity

Since only NPN invariant operations have been performed on the spectrum, the resultant spectrum can only be an NPN-equivalent function of the original spectrum. By simple book-keeping, the input permutations and phase assignments are easily obtained.

The overall complexity of the matching algorithm is, in the worst case, $O(n! \cdot 2^n)$ for an n variable functions. The most dominating operation is finding permutations of input ordering. Even though the problem is intractable, a few significant savings are possible. First, the worst case does not occur frequently. As indicated above, in most spectra, the number of distinct magnitudes in first order coefficients is greater than 2. Secondly, the algorithm dynamically traverses the possible permutations, and terminates as soon as a match is found. Most Boolean functions have some symmetry variables [7], therefore there will generally be more than one match in the set of input orderings. The detection of the first match is sufficient since the remaining matches are only symmetrical variants. Third, it is noted that most of the complexity lies in traversing 2^n spectrum coefficients. The regularity of the spectra size offers simple data structure (i.e. array or bit vector) and data objects (i.e. integers). The simplicity of computations lends to very fast computer operations (i.e. bitwise arithmetic operations and comparisons).

Circuit	Specter				Ceres			
	depth 3		depth 5		depth 3		depth 5	
	cost	time	cost	time	cost	time	cost	time
9symml	220	7.8	220	28.1	214	7.8	214	26.6
C1355	428	12.3	410	24.9	404	12.1	404	21.6
C1908	597	18.0	590	43.2	609	18.0	596	36.4
C2670	892	37.4	884	84.7	860	35.1	852	80.0
C3540	1245	51.3	1206	124.8	1240	52.3	1201	114.0
C432	218	7.0	218	20.7	218	6.9	218	18.9
C5315	2055	107.4	2049	206.7	2032	102.3	2005	185.4
C880	319	12.3	315	30.1	315	11.0	310	27.8
alu2	591	29.2	577	107.6	590	26.6	567	91.0
alu4	1036	56.0	1012	178.1	1023	50.9	1008	153.8
apex6	687	26.9	687	51.7	687	24.2	687	47.1
apex7	288	10.3	283	22.0	284	9.3	268	18.7
b1	14	0.6	14	1.2	14	0.5	14	1.0
cc	75	2.1	75	3.4	76	1.9	76	2.8
cht	313	12.7	253	27.8	313	11.4	222	25.6
CM163	50	1.5	48	2.8	50	1.3	48	2.7
cmb	53	1.9	49	6.7	51	1.8	50	5.0
cu	74	2.4	73	5.7	74	2.1	72	4.5
decod	48	1.2	48	1.2	48	1.1	48	1.1
example2	355	11.5	349	22.5	366	10	349	19
f51m	249	11.8	248	52.0	252	10.8	248	45.9
frg1	605	33.4	605	161.3	605	30.1	605	128
k2	2314	165.8	2318	502.1	2322	149.0	2326	292.3
lal	182	5.9	166	17.7	177	5.7	155	11.8
pair	1781	92.5	1644	183.6	1729	84.0	1576	144.5
rot	1154	68.2	1132	150.7	1139	65.8	1093	144.3
x1	1693	136.5	1684	432	1691	111.4	1659	338.8
x3	1406	84.6	1327	144.6	1342	62.2	1150	111.1
Total	18942	981.6	18484	2637.9	18725	905.6	18021	2099.7
	1.012	1.084	1.026	1.256	1.00	1.00	1.00	1.00

Table 1: LSI Library Mapping Results, *Specter* vs. *Ceres*

4 Results

The algorithm described above have been implemented as part of the *specter* technology mapper. A subset of the benchmarks from the 1991 MCNC suite have been mapped. The results are shown in Figure 1 for the LSI standard cell library, and in Figure 2 for the Actel library. The results mapped with spectral matching are listed under *Specter*.

Because *ceres* and *specter* only differ in the matching algorithm, the differences in results should not be significant. However, because the two matching algorithms return different ordering of inputs, the covering step may generate different results.

For the benchmarks listed, *specter* mappings cost more in both area and runtime. However, for LSI library, the difference is less than 3%. This is not significant, and can be attributed to implementation details. However, the runtimes for *specter* are much higher, as expected. At depth 3 [1], *specter* matching algorithm is fairly competitive. As the depth increases, the number of inputs to the candidate circuit increases as well. Since the spectral algorithm is exponentially sensitive to size of inputs, the runtime at higher depth is signifcantly slower. For depth of 5, *specter* is 25% slower than *ceres*. This illustrates the exponential nature of the algorithm.

For the Actel library, the difference in area is more significant. For a depth of 5, the resulting mapping cost differs by 6%. This again can be attributed to the different orderings returned by matching algorithms, and the nature of the library. Actel library elements are functionally more complex, often leading to more matches. Thus, selecting the one with best input ordering often can make large differences. The runtimes for *specter* are much more comparable at depth of 5, although for depth of 3, there is a 14% difference.

5 Conclusion and Acknowledgement

A first step towards applying spectral techniques to Boolean matching has been illustrated. In particular, a spectral matching algorithm based on the Hadamard transform for determining the NPN-equivalence of two functions is presented. We have shown that the algorithm offers significant pruning of the exponential search space, and uses fast operations and simple data structures. The results implemented in *specter* offer encouraging results.

The authors would like to thank Frédéric Mailhot for his insights and many helpful discussions. This research was sponsored by NSF-ARPA, under grant No. MIP 8719546 and, by AT&T and DEC jointly with NSF, under a PYI Award program. We acknowledge also support from ARPA, under contract No. J-FBI-89-101

References

[1] M. L. Dertouzos. *Threshold Logic: A Synthesis Approach.* M.I.T. Press, Cambridge, Massachusetts, 1965.

[1] For a definition of *depth*, see [7]

[2] E. Detjens and G. Gannot and R.L. Rudell and A. Sangiovanni-Vincentelli and A.R. Wang *Technology Mapping in MIS* In *Proceedings of ICCAD*, pages 116–119, Santa Clara, November 1987.

[3] C. R. Edwards. *Application of Rademacher-Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis. IEEE Transactions on Computers*, pages 48–62, January 1975.

[4] D. Gregory and K. Bartlett and A. de Geus and G. Hachtel SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic In *Proceedings of 23rd DAC*, pages 79–85, June 1986.

[5] S. L. Hurst, D. M. Miller, and J. C. Muzio. *Spectral Techniques in Digital Logic.* Academic Press, New York, New York, 1985.

[6] K. Keutzer DAGON: Technology Binding and Local Optimization by DAG Matching In *Proceedings of 24th DAC*, pages 341–347, June 1987.

[7] F. Mailhot *Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations* Ph.D. Dissertation, Stanford University, December 1991

[8] E. Pitty *A Critique of the GATEMAP Logic Synthesis System* In *Logic and Architecture Synthesis and Silicon Compilers*, pages 65–84.

Circuit	Specter				Ceres			
	depth 3		depth 5		depth 3		depth 5	
	cost	time	cost	time	cost	time	cost	time
9symml	107	8.0	106	25.7	97	7.3	96	24.3
C1355	180	12.3	180	22.8	178	11.9	178	23.0
C1908	317	17.4	304	37.5	281	15.9	274	37.5
C2670	410	34.4	396	72.0	403	31.2	372	68.0
C3540	699	50.2	665	105.9	649	45.7	624	100.5
C432	118	6.8	115	18.3	115	6.4	108	17.9
C5315	1019	93.5	1002	169.2	906	83.4	872	160.6
C880	200	11.3	190	24.6	195	10.4	181	23.1
alu2	336	27.8	331	79.2	335	23.8	321	74.9
alu4	576	49.7	566	132.9	576	44.5	552	126.3
apex6	409	24.5	407	52.2	401	21.9	398	40
apex7	154	9.5	152	18.4	149	8.4	141	17.0
b1	7	0.5	7	0.9	6	0.4	6	0.9
cc	41	2.2	40	3.0	34	1.9	33	0.8
cht	161	11.5	132	23.4	161	10.5	125	22.3
CM163	25	1.4	21	2.9	24	1.3	20	2.8
cmb	27	2.0	27	4.6	26	1.7	26	4.3
cu	34	2.3	34	4.3	34	2.1	32	4.0
decod	26	2.1	26	1.1	26	1.0	26	1.0
example2	181	10.6	174	18.6	179	9.7	164	17.8
f51m	136	11.6	135	37.7	137	10.3	133	36.6
frg1	283	32.8	279	4.7	283	28.7	280	100.9
k2	1182	155.5	1182	262.9	1179	135.3	1179	238.8
lal	96	6.1	91	11.8	95	5.5	77	10.4
pair	924	91.0	823	144	914	78.2	761	130.9
rot	596	61.2	570	135.4	576	52.3	540	124.9
x1	853	127.3	837	303.4	844	105.7	829	278
x3	721	75.4	669	130.4	671	64.5	560	111.2
Total	9818	938.9	9461	1847.8	9474	819.9	8908	1798.6
	1.038	1.145	1.062	1.027	1.00	1.00	1.00	1.00

Table 2: Actel Library Mapping Results, *Specter* vs. *Ceres*