Fast Boolean Matching for LUT Structures

Alan Mishchenko **Satrajit Chatterjee** **Robert Brayton**

Department of EECS, University of California, Berkeley

{alanmi, satrajit, brayton}@eecs.berkeley.edu

Abstract

This paper addresses the problem of mapping a completely-specified Boolean function of 16 or fewer variables, into a network of K-input lookup-tables (*K-LUTs*) where $3 \le K \le 6$. *The proposed algorithm is based on cofactoring and disjoint-support decomposition and is complete* (i.e. *capable of finding the smallest network of K-LUTs needed to implement the function*). *The algorithm is several orders of magnitude faster than previous work that relies on BDDs for functional decomposition or on Boolean Satisfiability for FPGA architecture evaluation since it exploits the Boolean structure of the function being mapped and uses truth-tables to represent functions. The algorithm also admits an incomplete implementation where exact optimality is sacrificed for run-time. The incomplete implementation was used for fast area-oriented resynthesis of mapped networks with promising results. The proposed resynthesis preserves depth while reducing area by 7.1% after state-of-the-art FPGA mapping and by 5.4% after another type of high-effort area-oriented resynthesis. This indicates that the resynthesis based on Boolean matching is largely orthogonal to prior techniques and can be utilized independently or on top of the existing approaches for additional area reduction.*

1 Introduction

We consider the problem of matching completely-specified Boolean functions to *LUT structures,* i.e. networks of K-input lookup-tables (K-LUTs). In this paper, we assume that K is fixed (typically $3 \le K \le 6$). Figures 1.1 and 1.2 show examples of LUT structures: a configuration with two Stratix II programmable logic blocks which can implement a subset of Boolean functions of up to 11 inputs (Figure 1.1); and a combination of four 3-LUTs used in Actel ProASIC3 devices which can implement some Boolean functions up to 9 inputs (Figure 1.2).

The matching problem for LUT structures described above has several applications: evaluating the expressive power of FPGA architectures, mapping designs into these architectures, and re-synthesizing mapped designs to reduce area, delay, etc.

The problem of Boolean matching to LUT structures can take two forms. In the first form, we are given a network *N* of K-LUTs and a function *f* and asked if *f* can be implemented by *N*. In the second form, we are given a function *f* and are asked to find the smallest network of LUTs that can implement *f.* In this paper we develop an algorithm for solving both forms of the Boolean matching problem, but present an application (area-oriented resynthesis) only for the second form.

For the first form of the problem, we may classify Boolean matchers into two types. A *complete* matcher always finds a solution if it exists. An *incomplete* matcher may not be able to find a solution in some cases where a complete matcher finds one. On the other hand, the runtime of an incomplete matcher is typically more affordable and therefore incomplete matchers are also useful in practical applications. The notions of complete and incomplete matchers also extend to the second form of the problem in a natural way. If the matcher is guaranteed to return the smallest LUT structure then it is complete; otherwise it is incomplete.

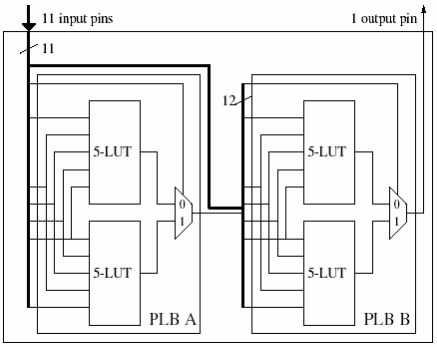

Figure 1.1. The LUT structure in Altera Stratix II devices [3].

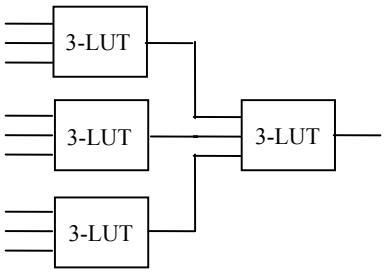

Figure 1.2. The LUT structure in Actel ProASIC3 devices [2].

The contributions of this paper are three-fold:

- A new complete algorithm is proposed for efficient Boolean matching into LUT structures. A heuristic version is also developed that trades completeness for speed.
- A new area-oriented resynthesis engine is developed based on the heuristic version of the Boolean matcher.
- Experimental evaluation of the proposed resynthesis is performed to show that (a) the proposed Boolean matching is reasonably fast and efficient, and (b) the resynthesis is scalable and complementary to other similar methods [28].

The rest of the paper is organized as follows. Section 2 reviews previous work. Section 3 gives a motivating example. Section 4 introduces terminology and background. Section 5 describes the details of the new algorithm. Section 6 describes a resynthesis framework based on the new algorithm. Section 7 reports experimental results. Section 8 concludes the paper.

2 Comparison with previous work

Boolean matchers that are based on different notions can be compared using the following criteria:

- An average *runtime* needed to match a given K-variable function with a given N-variable LUT structure (K ≤ N).
- An average *success rate*. This is the percentage of functions successfully matched, out of the total number of functions, for which a match exists.

A number of approaches have been developed for solving the Boolean matching problem:

(1) *Structural approach*. [10][27] The structural LUT-based technology mappers fall into this category. This approach is incomplete because it requires the subject graph of a Boolean function to match the LUT structure. It is fast but has a low success rate. The actual rate may strongly depend on the function and the structure used. It our experiments with 16-input functions matched into structures composed of 6-LUTs, the success rate was about 30%.

(2) *SAT-based approach*. [19][33][22][16] This complete method is guaranteed to find a match if it exists. It is also flexible because it can be easily customized to different LUT structures. However, it is time-consuming and takes seconds or minutes to solve 10-variable instances. When the problem is hard and runtime limit is used, this method becomes incomplete with the success rate depending on the problem type. For example, [22] reports a success rate of 50% for 10-variable functions with a 60 second timeout.

(3) *NPN-equivalence-based approach*. [13][7][1][8]. This potentially complete method pre-computes all NPN-classes of functions that can be implemented using the given architecture. Although this method is relatively fast, it does not scale well for functions above 9-12 inputs because of the large number of NPN classes. It is restricted to a particular LUT structure and may not be feasible for large/complex architectures with too many NPN-classes.

(4) *Functional approach.* [17][35][23][24] This approach attempts to apply a Boolean decomposition algorithm to break the function down into K-input blocks. The runtime strongly depends on the number of inputs of the function, the type of decomposition used, and how the decomposability check is implemented. The completeness is typically compromised by the heuristic nature of bound-set selection [17] or fixing variable order in the BDD [35] when looking for feasible decompositions. Exhaustively exploring all bound-sets, as proposed in [24], can only be done for relatively small functions (up to 12 inputs).

The Boolean matching algorithm proposed in this paper belongs to the last category. It uses properties of Boolean functions, such as disjoint-support decomposition, to guide what K-input blocks to extract. The completeness of this method is guaranteed by Theorem 1 presented in Section 5.1. In practice the completeness requires performing disjoint-support decomposition for a large number of cofactors of the function. Therefore, our current implementation limits the number of cofactors considered, which makes the method fast but incomplete. Experiments show that the cases when the given method does not find a match with a given structure, or fails to map a function into fewest LUTs, are rare.

3 A motivating example

Consider mapping a 4:1 MUX into two 4-LUTs. The problem cannot be solved using a structural mapper if the starting logic structure is represented as shown in Figure 3.1 (left). In this case, the only feasible structural mapping takes three 4-LUTs, each containing a 2:1 MUX, leaving one input of each 4-LUT unused.

To achieve a more compact mapping, cofactoring of the original function F is performed w.r.t. variable x. This leads to cofactors $F_{x=0} = ya' + yb$ and $F_{x=1} = yc' + yd$, sharing variable y. Now, 4-input block with the output signal z is created, as shown in Figure 3.1 (right): $z = x'F_{x=0} + xy$. The output of this block can be either $F_{x=0}$ (when $x = 0$) or y (when $x = 1$). After creating block z, the rest of the function can be packed into the second 4-input block depending on x, z, c, and d. As a result, function F of a 4:1 MUX is realized by a LUT structure composed of two 4-LUTs.

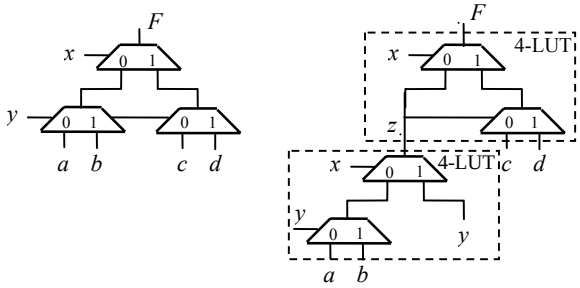

Figure 3.1. Mapping 4:1 MUX into two 4-LUTs.

This example shows that cofactoring the function and grouping parts of the cofactors into decomposable blocks can result in logic restructuring, which has a match with the given LUT structure. A theoretical result and an algorithm are presented below (Sections 5 and 6, respectively) to find similar matching whenever possible.

4 Background

Boolean network

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network and circuit are used interchangeably in this paper.

A node n has zero or more *fanins,* i.e. nodes that are driving n, and zero or more *fanouts*, i.e. nodes driven by n. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs in this paper. It is assumed that each node has a unique integer called its *node ID.*

A *fanin* (*fanout*) *cone* of node n is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node n is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through n. Informally, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed or substituted, its MFFC can be removed.

Structural cuts

A *cut* C of node n, called *root*, is a set of nodes of the network, called *leaves*, such that each path from a PI to n passes through at least one leaf. A *trivial cut* of node n is the cut $\{n\}$ composed of the node itself. A non-trivial cut *covers* all the nodes found on the path from the root to the leaves, including the root and excluding the leaves. A trivial cut does not cover any nodes. A cut is *K-feasible* if the number of nodes in it does not exceed K. A cut is said to be *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

Technology mapping

Technology mapping has a goal of representing a given network as an interconnection of logic blocks, each of which is implemented using a particular technology. In this paper, we deal with mapping into K-LUTs, which are devices capable of implementing logic functions whose support does not exceed K.

A Boolean network is *K-bounded* if its nodes are *K-feasible*, that is, the number of their fanins does not exceed *K*. A *subject graph* is a *K*-bounded network used for technology mapping. Any combinational network can be transformed into an AND-INV graph (AIG), composed of two-input ANDs and inverters. In the rest of the paper, the subject graph is assumed to be an AIG.

The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path length but the PIs are not. The network *depth* is the largest level of an internal node in the network. The depth and area of an FPGA mapping are measured by the depth of the resulting LUT network and the number of LUTs in it.

A *mapping* assigns one K-feasible cut, called the *representative cut*, to each non-PI node of the subject graph. The procedure also computes and incrementally updates a subset of nodes whose representative cuts cover all non-PI nodes in the graph. These nodes are said to be *used* in the mapping. During mapping, the representative cuts of the nodes are updated. Each such update may change the set of used nodes. These changes may propagate recursively from the node towards the PIs. The *area* of the mapping is the number of nodes used in the mapping.

Disjoint-support decomposition

A completely-specified Boolean function *F essentially depends* on a variable iff the value of the function differs for at least one pair of input combinations, in which the value of this variable is different while the values of other variables are the same. The *support* of *F* is the set of all variables, on which function *F* depends essentially. The supports of two functions are *disjoint* if they do not contain common variables. The supports of a number of functions are *disjoint* if their supports are pair-wise disjoint.

A *decomposition* of a completely-specified Boolean function is a Boolean network that is functionally equivalent to the function. A *disjoint-support decomposition* (DSD) is a decomposition, in which the nodes of the resulting network are represented by functions with disjoint support. A DSD is called *DSD of the finest granularity*, if each of the nodes of the resulting network cannot be further decomposed by DSD.

The internal nodes of the DSD network can be of three types: AND, XOR, and PRIME. The AND and XOR nodes may have any number of inputs, while the PRIME nodes represent components without DSD whose support size is three or more. For example, the 2:1 MUX is a prime node with three inputs.

Theorem [4]. For a completely-specified Boolean function, DSD of the finest granularity is canonical, that is, there is only one Boolean network (up to the complementation of inputs and outputs of the nodes) representing the function, with the property that none of its nodes can be further decomposed using DSD.

There are several efficient algorithms for computing the DSD of the finest granularity of a multi-output completely-specified Boolean function [6][34][21]. Our implementation follows [6] but uses truth tables instead of BDD to manipulate Boolean functions.

5 Boolean matching

In this section, we discuss the theory and algorithms for Boolean matching.

The problem solved by the Boolean matching in this paper is: Given a Boolean function, can it be decomposed into no more than the given number (N) of K-feasible blocks? Each of the decomposed blocks is realized as a K-LUT, resulting in a LUT structure composed of no more than N K-LUTs. By trying different values of N, we can find the minimum N, for which decomposition exists, which give us an implementation with the fewest K-LUTs. Matching into predefined LUT-structures is a closely related problem that can be solved similarly.

5.1 Theory

The approach to decomposition described in this paper relies on the previous work in constructive decomposition [17], also known as Ashenhurst-Curtis decomposition [4][12]. This approach was extended to work with incompletely-specified and multi-valued functions and relations [31][15]. In this approach, the function is transformed into a Boolean network by iteratively extracting one K-feasible block at a time. The process converges when the function after decomposition (called the *composition function*) is simpler than the original function in some sense. The set of input variables included in the K-feasible block that is decomposed out while reducing the support of the composition function is called *support-reducing bound-set*.

Several previous approaches are known to solve the problem of bound-set selection [15][17][35][24]. The solution proposed in this paper is the closest to [23] with some ideas coming from [32].

Unlike the previous formulations of constructive decomposition for Boolean functions, the current work requires that the decomposed block has only one output but allows it to share variables with the composition function. The question is what variables should be unique for the block and what variables should be shared, so that the total support of the block does not exceed K, while the number of unique variables in it is maximized. This guarantees the maximum support-reduction of the decomposition with the given variables as a bound-set.

The proposed approach finds the bound-set with the largest support reduction. The key insight is summarized in Theorem 1 and illustrated in Figure 5.1.

Theorem 1: Function *F* has decomposition of the type $F = H(a, b, D(b, c))$, where *a* and *c* are variable of only *H* and *D*, respectively, and *b* are common variables of *H* and *D*, if and only if all $2^{|b|}$ cofactors of *F* w.r.t. variables *b* have DSD structures, in which variables *c* are decomposable as separate blocks.

Consider an illustration of Theorem 1. Suppose there exists an implementation of function *F* using two blocks, *H* and *D*, as shown in Figure 5.1 (left). Let function *F* have no DSD and variable *b* be the only common variable of *H* and *D*. Cofactoring *F* w.r.t. *b* results in the structures of the cofactors $F_{b=0}$ and $F_{b=1}$, shown in Figure 5.1 (right). With variable *b* replaced by a constant in the support of *H* and *D*, both cofactors have DSDs with bound-set *c* and remaining variables *a*.

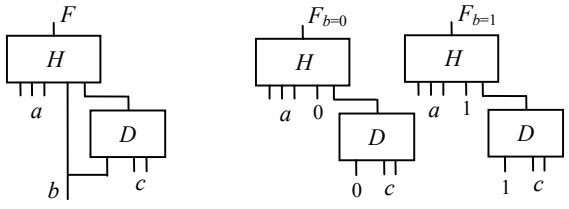

Figure 5.1. Illustration of the use of DSD for the cofactors of the function to find support-reducing decompositions.

Going from the cofactor DSDs on the right of Figure 5.1 to the decomposition structure on the left can be done by detecting bound-set *c* common to both cofactor DSDs.

Consider now the motivating example in Section 3. The original function shown in Figure 3.1 (left) has no DSD. However, after cofactoring w.r.t. variable *x*, both cofactors have DSDs. The bound-set {*a, b, y*} is common to both cofators. When combined with the cofactoring variable *x*, this bound-set leads to the decomposition composed of two 4-LUTs in Figure 3.1 (right).

When the search for common bound-sets of the cofactor DSDs is exhaustive, and cofactoring is tried w.r.t. all variables in the support of *F*, the proposed approach is complete in the following sense: for the given Boolean function, it finds the decomposition with the minimum number (*N*) of K-input blocks, if such decomposition exists.

In our current implementation, we make heuristic trade-offs between the quality of decomposition found and the runtime spent in exploring bound-sets. The limit is imposed on (a) the number of variables used for cofactoring, (b) the number of different variable combinations tried. Our experiments show that for many problems, the proposed algorithm finds the minimum solution.

5.2 Algorithm

The pseudo-codes in this subsection show how the DSD structures of the cofactors can be exploited to compute a bound-set which leads to the maximum support reduction during constructive decomposition.

```
varset findSupportReducingBoundSet( function F, int K )
{
    // derive DSD for the function
    DSDtree Tree = performDSD( F );
    // find K-feasible bound-sets of the tree
    varset BSets[0] = findKFeasibleBoundSets( F, Tree, K );
    // check if a good bound-set is already found
    if ( BSets[0] contains bound-set B of size K )
        return B;
    if ( BSets[0] contains bound-set B of size K -1 )
        return B;
    // cofactor F w.r.t. sets of variables and look for the largest
    // support-reducing bound-set shared by all the cofactors
    for ( int V = 1; V ≤ K – 2; V++ ) {
      // find the set including V cofactoring variables
      varset cofvars = findCofactoringVarsForDSD( F, V );
      // derive DSD trees for the cofactors and compute
      // common K-feasible bound-sets for all the trees
      set of varsets BSets[V] = {∅};
      for each cofactor Cof of function F w.r.t. cofvars {
        DSDtree Tree = performDSD( Cof );
        set of varsets BSetsC =computeBoundSets(Cof, Tree, K-V);
        BSets[V] = mergeSets( BSets[V], BSetsC, K-V );
      }
      // check if at least one good bound-set is already found
      if ( BSets[V] contains bound-set B of size K-V )
        return B;
      // check for comparable bound-sets with fewer cofactoring
      // variables, before trying to use more shared variables
      for ( int M = 0; M ≤ V; M++ )
        if ( BSets[M] contains bound-set B of size K-V-1 )
          return B;
    }
    return NONE;
}
```

Figure 5.2.1. Computing a good support-reducing bound-set.

The procedure **findSupportReducingBoundSet** in Figure 5.2.1 takes a completely-specified function *F* and the limit *K* on the suppose size of the decomposed block. It returns a good bound-set, that is, a bound-set leading to the decomposition with a maximal support-reduction. If support-reducing decomposition does not exist, the procedure returns NONE.

First, the procedure derives the DSD tree of the function itself. The tree is used to compute the set of all feasible bound-sets whose size does not exceed *K*. Bound-sets of larger size are not interesting because they cannot be implemented using K-LUTs. For each of the bound-sets found, decomposition with a single output and no shared variables is possible. If a bound-set of size K exists, it is returned. If such bound-set does not exist, the second best would be to have a bound-set of size K-1. If such bound-set also does not exist (for example, when the function has no DSD), the computation enters a loop, in which cofactoring of the function w.r.t. several variables is tried, and common support-reducing bound-sets of the cofactors are explored.

When the loop is entered, cofactoring w.r.t. one variable is tried first. If two cofactors of the function have DSDs with a common bound-set of size K-1, it is returned. In this case, although the decomposed block has K variables, only K-1 variables are reduced because the cofactoring variable is shared. If there is no common bound-set of size K-1, the next best outcome is one of the three: (i) there is a bound-set of size K-2 of the original function, (ii) there is a common bound-set of size K-2 of the two cofactors w.r.t. the cofactoring variable, or (iii) there is a common bound-set of size K-2 of the four cofactors w.r.t. two variables. The loop over *M* at the bottom of the pseudo-code in Figure 5.2.1 is testing for outcomes (i) and (ii). If they are impossible, *V* is incremented and the next iteration of the loop is performed, which is the test for the outcome (iii).

In the next iteration over *V*, cofactoring w.r.t. two variables is attempted and four resulting cofactors are searched for common bound-sets. The process is continued until a bound-set is found, or the cofactoring w.r.t. K-2 variables is tried without success. When *V* exceeds K-2 (say, *V* is K-1), the decomposition is not support-reducing, because the composition function depends on shared K-1 variables plus the output of the decomposed block. In other words, the decomposition takes away K variables from the composition function and returns K variables. In this case, NONE is returned, indicating that there no support-reducing decomposition.

Example. Consider decomposition of function *F* of the 4:1 MUX shown in Figure 3.1 (left). Assume K = 4. This function does not have DSD, that is, its DSD is composed of one prime block. The set of K-feasible bound-sets is trivial in this case: { {∅}, {*a*}, {*b*}, {*c*}, {*d*}, {*x*}, {*y*} }. Clearly, none of these bound-set have size K or K-1. The above procedure enters the loop with V = 1. Suppose *x* is chosen as the cofactoring variable. The cofactors are $F_{x=0} = ya' + yb$ and $F_{x=1} = yc' + yd$. The K-1-feasible bound-sets are {{∅}, {*a*}, {*b*}, {*y*}, {*a, b, y*}}, and {{∅}, {*c*}, {*d*}, {*y*}, {*c, d, y*}}. A common bound-set {*a, b, y*} of size K-1 exists. The loop terminates and this bound-set is returned, resulting in the decomposition shown in Figure 3.2 (right).

Figure 5.2.2 shows **findCofactoringVarsForDSD**, which computes the set of *N* cofactoring variables of the function to be decomposed. The variables are heuristically selected so that the cofactors have as small non-DSD components as possible. Ideally, all cofactors w.r.t. the selected set of variables should be fully decomposable using DSD because this maximizes the chances of finding good bound-set. However, finding such set of variables

may be hard or impossible, given the restriction on the number of the cofactoring variables.

It was found that cofactoring a function w.r.t. all variable subsets and examining decomposability of the cofactors is slow even for the efficient truth-table-based implementation. As a result, a heuristic is used, which selects variables, one at a time, and adds them to the resulting set. A new variable is selected so that the sum total of support sizes of the cofactors w.r.t. all currently selected variable is minimized.

```
varset findCofactoringVarsForDSD( function F, int N )
{
    varset ResVars = ∅;
    set of functions Cofactors = {F};
    varset CandVars = supp(F);
    // find variables that minimize the sum total of cofactor supports
    for ( int M = 1; M ≤ N; M++ ) {
        var v = findVarMinSupportSizes( Cofactors, CandVars );
        Cofactors = cofactorMultipleFunctions( Cofactors, v );
        CandVars = CandVars \ v;
        ResVars = ResVars ∪ v;
    }
    return ResVars;
}
```

Figure 5.2.2. Computing a set of *N* cofactoring variables.

Figure 5.2.3 shows procedure **findKFeasibleBoundSets**, which takes the decomposition tree of a function and integer *K* and computes the set of all *K*-feasible bound-sets leading to a simple decomposition of the function, that is, the decomposition with one single-output block having no more than K inputs, without shared variables.

```
set of varsets computeBoundSets_rec( DSDnode Node, int K )
{
    // consider terminal node of the tree
    if ( nodeType(Node) == VAR ) {
        return supp(Node);
    }
    // collect K-feasible variable sets derived for the fanins
    set of varsets Result = ∅;
    for each Fanin of Node
        Result = Result ∪ computeBoundSets_rec( Fanin, K );
    // consider AND and XOR decomposition
    if (nodeType(Node) == AND or nodeType(Node) == XOR ) {
        // create combinations of the fanins supports
        for each subset S of fanins of Node
            if ( |supp( S )| ≤ K )
                Result = Result ∪ supp( S );
    }
    // consider non-decomposable node
    if ( nodeType(Node) == PRIME ) {
        if ( supp(Node) ≤ K )
            Result = Result ∪ supp( Node );
    }
    return Result;
}
set of varsets computeBoundSets( F, Tree, K )
{
    // consider trivial cases of DSD tree
    if ( |supp(F)| = 0 )
        return {{∅}};
    if ( |supp(F)| = 1 )
        return {{∅}, supp(F)};
    return {∅} ∪ computeBoundSets_rec( rootNode(Tree), K );
}
```

Figure 5.2.3. Computing K-feasible bound-sets of the DSD tree.

This procedure processes nodes of the DSD-tree differently, depending on their type. The terminal node of the tree represents an input variable. In this case, the support of the node is returned. For the other types of nodes, the procedure collects the K-feasible supports computed for each fanin. If the node is a multi-input AND or XOR, its fanins are commutative. As a result, any subset of fanins can be decomposed out. This is why subsets of the fanins are considered, and if the total support of a subset of the fanins is K-feasible, it is added to the set of K-feasible bound-sets. Finally, if the node is prime, its fanins cannot be grouped together to produce a single-output decomposition block. This is why only the support of the node itself is added to the result, provided it is K-feasible. Before the computed set of bound-sets is returned to the user, an empty set is added to it. This is needed for merging of the sets of K-feasible bound-sets.

Figure 5.2.4 shows the procedure **boundSetsMerge**, which takes the sets of bound-sets of two cofactors and returning the set of bount-sets that are common to both cofactors. It considers all bound-set pairs and computes their unions. Next, it removes those bound-sets whose size exceeds K and those that overlap with the remaining variables in the support of the cofactors. Such bound-sets, if used, could duplicate a variable between the decomposed block and the composition function. Meanwhile, currently duplication is restricted only to the cofactoring variables.

```
set of varsets boundSetsMerge( sets of varsets BSet1, BSet2, int K )
{
    set of varsets Result = ∅;
    for each bound-set B1 in BSet1 {
        for each bound-set B2 in BSet2 {
            B = B1 ∪ B2;
            if ( | B | > K )
                continue;
            if ( (~B1 ∩ B) or (B1 ∩ ~ B) )
                continue;
            Result = Result ∪ B;
        }
    }
    return Result;
}
```

Figure 5.2.4. Merging two sets of K-feasible bound-sets.

6 Resynthesis framework

The proposed approach to Boolean matching is used in area-oriented resynthesis that is applied to a network after technology mapping for FPGAs. The nodes of the network have K inputs or less and will be realized by K-LUTs.

The resynthesis framework presented in this paper has three independent components:

- Cut computation (Section 6.1)
- Finding support reduction bound-set (Section 5)
- Decomposition and network update (Section 6.2)

6.1 Cut computation

The cut computation for resynthesis differs from the cut enumeration for technology mapping in several ways:

- Mapping is applied to an AIG, while resynthesis is applied to a mapped network.
- Mapping considers nodes in a topological order, while resynthesis may be applied to selected nodes, for example, nodes on a critical path, or nodes whose neighbors have changed in the last pass of resynthesis.
- During mapping, the size of a cut's MFFC is not important (except during some types of area recovery), while in

resynthesis the size of a cut's MFFC is the main criterion to judge how many LUTs may be saved after resynthesis.

- Mapping for K-LUTs requires cuts of size K whereas for an efficient resynthesis of a K-LUT network, larger cuts need to be computed (typically up to 16 inputs).

Given these observations, complete or partial cut enumeration used extensively in technology mapping is not well-suited for resynthesis. Resynthesis requires a different cut computation strategy that is top-down: cuts are computed separately for each node starting with the trivial cut of the node and new cuts are obtained by expanding existing cuts towards primary inputs.

The pseudo-code of the cut computation is given in Figure 6.1. The procedure takes the root node (*root*), for which the cuts are being computed, the limit on the cut size (*N*), the limit on the number of cuts (*M*), and the limit on the number of nodes duplicated when the computed cut is used for re-synthesized (*S*). The last number is the maximum number of nodes that can be covered by a cut, which are not in the MFFC of the given node. Obviously, these nodes will be duplicated when the cut's function is expressed in terms of the cut leaves, using a set of new K-feasible nodes after decomposition.

For each cut computed, two sets of nodes are stored: the set of leaves and the set of nodes covered by the cut (these are the nodes found on the paths between the leaf nodes and the root, excluding the leaves and including the root). Additionally, each cut has a counter of covered nodes that are not in the MFFC of the root. The value of this counter is returned by procedure **numDups**.

```
cutset cutComputationByExpansion ( root, N, M, S )
{
    markMffcNode( root );
    cutset = { {node} };
    for each cut in cutset {
        for each leaf of cut
            if ( !nodeIsPrimaryInput( leaf ) )
                expandCutUsingLeaf( cut, leaf, N, S, cutset );
    }
    filterCutsWithDSDstructure( cutset );
    if ( |cutset| > M ) {
        sortCutsByWeight( cutset );
        cutset = selectCutsWithLargerWeight( cutset, M );
    }
    return cutset;
}

expandCutUsingLeaf ( cut, leaf, N, S, cutset )
{
    // check if the cut is duplication-feasible
    if ( !nodeBelongsToMffc(leaf) && numDups(cut) == S )
        return;
    // derive the new cut
    cut_new = (cut \ leaf ) ∪ nodeFanins( leaf );
    // check if the cut is N-feasible
    if ( numLeaves( cut_new ) > N )
        return;
    // check if cut_new is equal to or dominated by any cut in cutset
    // (also, remove the cuts in cutset that are contained in cut_new)
    if ( cutsetFilter( cutset, cut_new ) )
        return;
    // add cut_new to cutset to be expanded later
    cutset = cutset ∪ cut_new;
}
```

Figure 6.1. Cut computation for resynthesis.

Procedure **expandCutUsingLeaf** tries to expand the cut by moving a leaf to the set of covered nodes and adding the leaf's fanins to the set of leaves. If the leaf belongs to the MFFC of the root and the number of duplicated nodes of the cut has already saturated, the new cut is not constructed. If the resulting cut has more leaves than is allowed by the limit, the new cut is not used. If the cut is equal or dominated by any of the previously computed cuts, it is also skipped. Finally, if none of the above conditions holds, the cut is appended to the resulting cutset. Later in the cut computation, this cut, too, is used to derive other cuts.

The above procedure converges because of the limits on the cut size (*N*) and the number of duplicated nodes (*S*), resulting in a set of cuts that are potentially useful for resynthesis. A cut is actually used for resynthesis if both of the following conditions are met: (a) the cut does not have a DSD-structure and (b) the cut has a high enough weight. These conditions are described below.

Cuts with a DSD structure

Some cuts may be not useful for resynthesis because the network structure covered by these cuts cannot be compacted. These are the cuts covering a part of the logic structure of the current network, which includes one or more nodes that are disjoint-support-decomposable from the rest of the structure. For example, Figure 6.2 shows a cut {*a, b, c, d*} of node *n*. This cut covers node *m* whose fanins, *c* and *d*, are disjoint-support w.r.t. the rest of the covered network structure depending on *a* and *b*.

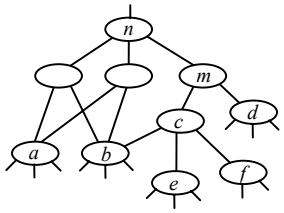

Figure 6.2. Illustration of a cut with DSD-structure.

The cuts with a DSD-structure can be efficiently identified by a dedicated procedure. Such cuts can be skipped when attempting resynthesis, but they cannot be left out during the cut computation because expanding them can lead to other useful cuts. For the example in Figure 6.2, expanding cut {*a, b, c, d*} w.r.t. leaf *c* leads to a cut {*a, b, d, e, f*}, which does not have a DSD structure.

Cut weight

Some cuts should be skipped during resynthesis because they have little or no potential for improvement and should not be tried, given a tight runtime budget. To filter out useless cuts and prioritize other cuts, the notion of *cut weight* is introduced. The weight is computed for all cuts and only a given number (*M*) of cuts with high enough weight is selected for resynthesis.

The weight of a cut is defined as follows:

cutWeight(c) = [**numCovered**(c)– **numDups**(c)]/**numLuts**(c),
where **numLuts**(c) = ceiling[(**numLeaves**(c)-1)/(K-1)].

The procedures **numCovered** and **numDups** return the total number of covered nodes and the number of covered nodes not belonging to the MFFC of the root, respectively. Procedure **numLeaves** return the number of cut leaves. Finally, **numLuts** computes the minimum number of K-LUTs needed to implement the cut of the given size, provided that the cut's function has a favorable decomposition. Procedure **numLuts** is the inverse w.r.t. the parameter *N* of the formula for the number of free inputs, *V*, of the structure composed of *N* K-input LUTs: $V = N * (K-1) + 1$.

Intuitively, the weight of a cut shows how likely the cut will be useful in resynthesis: the more nodes it saves and the less LUTs may be needed to realize it, the better the cut. The cuts whose

weights are less or equal than 1.0 can be skipped because they give no improvement. If resynthesis with zero-cost replacements is allowed, only the cuts with weight less than one are skipped.

6.2 Decomposition and network update

Figure 6.2 gives an overall pseudo-code of the proposed approach to resynthesis:

```
resynthesisByLutPacking( network, parameters )
{
    // apply resynthesis to each node
    for each node of network in topological order {
        cutset = cutComputationByExpansion( node, parameters );
        for each cut in cutset in decreasing order of cut weight {
            // derive function of the cut as a truth table
            F = cutComputeFunction( cut );
            // iteratively decompose the function
            while ( suppSize(F) > K ) {
                boundset = findSupportReducingBoundSet( F, K );
                if ( boundset == NONE )
                    break;
                F = functionDecompose( F, boundset );
            }
            // if F is decomposed, update and move to the next node
            if ( boundset != NONE ) {
                networkUpdate( network );
                break;
            }
        }
    }
}
```

Figure 6.2. Overall pseudo-code of the proposed resynthesis.

6.3 Additional details

In this subsection, we briefly present several aspects of the proposed algorithm that were not described above to keep the discussion simple.

Using timing information to filter candidate bound-sets

The timing information, which can be taken into account by the algorithm, includes the arrival times of the leaves of the cut and the required time of the root. During decomposition, a lower-bound on the root's required time is computed and further decomposition is not performed if this bound exceeds the required time. In this case, the resynthesis algorithm moves on to the next cut of the root node. If the node has no more cuts, the algorithm moves to the next node in the topological order.

The use of the timing information prunes the search space and therefore leads to faster resynthesis. However, the gain in area may be limited due to skipping some of the resynthesis opportunities as not feasible under the timing constraints.

Restricting bound-sets for balanced decompositions

Experiments have shown that allowing all variables in the bound-sets may lead to unbalanced decompositions. Therefore, the variables that are allowed to be in the bound-sets are restricted to those that are in the transitive fanins cones of the prime blocks. If the DSD trees of the cofactors do not have prime blocks, all variables are allowed to be in the bound-sets.

Opportunistic MUX-decomposition

To further improve quality and runtime, another type of decomposition is considered along with the DSD-based decomposition presented in Section 5. This type of decomposition attempts to cofactor the function w.r.t. every variable and checks the sizes of cofactors. If the smaller of the two cofactors has support size less than K-2 and the larger one has size less than K, the function can be implemented using two K-LUTs, one of which subsumes the smaller block together with the MUX, while another is used as the larger block. This type of decomposition can often be found faster than a comparable decomposition based on DSD.

7 Experimental results

The proposed algorithm is implemented in ABC [5] as command *lutpack*. The experiments targeting 6-input LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The resulting networks were all verified using the combinational equivalence checker in ABC (command *cec*) [26].

The following ABC commands are included in the scripts used to collect the experimental results targeting area minimization:

- *resyn* is a logic synthesis script that performs 5 iterations of AIG rewriting [25]
- *resyn2* is a logic synthesis script that performs 10 iterations of AIG rewriting, which are more diverse than those of *resyn*
- *choice* is a logic synthesis script that allows for accumulation of structural choices; *choice* runs *resyn* followed by *resyn2* and collects three snapshots of the current network: the original, the final, and the intermediate one saved after *resyn*
- *if* is an efficient FPGA mapper with priority cuts [29], fine-tuned area recovery, and the capacity to use subject graph with structural choices (the mapper was run with the following settings: at most 12 6-input priority cuts are stored at each node; five iterations of area recovery are performed, three with area flow and two with exact local area)
- *imfs* is an area-oriented resynthesis engine for FPGA [28].
- *lutpack* is the new resynthesis engine described in this paper.

The benchmarks used in this experiment are 20 large public benchmarks from MCNC and ISCAS'89 suites used in previous work on FPGA mapping [20][10][27]. (In the above set, circuit s298 was replaced by i10 because it contains only 24 6-LUTs.)

Table 1 shows five experimental runs:

- Section "Baseline" corresponds to a typical run of tech-independent synthesis followed by default technology mapping (*resyn; resyn2; if*)
- Section "Choices" corresponds to four iterations of mapping with structural choices (*choice; if*).
- Section "Lutpack" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resynthesis (*choice; if*; *lutpack*), followed by several additional iterations of *lutpack*.
- Section "Imfs" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resynthesis (*choice; if*; *imfs*).
- Section "Imfs+Lutpack" corresponds to four iterations of mapping with structural choices, interleaved with the proposed resynthesis (*choice; if*; *imfs*), followed by several additional iterations of *lutpack*.

Table 1 lists the number of primary inputs (column "PIs"), primary outputs (column "POs"), registers (column "Reg"), area calculated as the number of 6-LUTs (columns "LUT") and delay calculated as the depth of the 6-LUT network (columns "Level"). The ratios in the tables are the ratios of geometric averages of values reported in the columns.

Table 1 demonstrates that iterative mapping with structural choices (Section "Choices") reduces area and delay on average by 5.5% and 8.1%, respectively, compared to the baseline synthesis and mapping (Section "Baseline"). We conjecture that this

improvement is due to repeated re-computation of structural choices by applying logic synthesis to the previously mapped network. When the resulting subject graph with choices is mapped again, those logic structures tend to be selected that offer an improvement, compared to the previous run of mapping. Several iterations of this evolutionary process lead to logic restructuring that is favorable for the selected LUT size and delay constraints.

When the proposed resynthesis is interleaved with mapping with structural choices (Section "Lutpack"), it achieves additional average reductions of 7.1% and 1.1% in area and delay, respectively, compared to mapping with structural choices alone (Section "Choices"). This improvement is likely because the proposed resynthesis allows for an even deeper restructuring of the subject graph that is favorable for area minimization. Applying logic synthesis to further improve the results of resynthesis, recording choices, running mapping with choices, and iterating this leads to an even better logic restructuring.

Section "Imfs" shows the result of interleaving mapping with structural choices and runs of another resynthesis engine *imfs*. Substantial reductions are achieved after *imfs* for some benchmarks, compared to mapping with structural choices. The large reductions are because powerful Boolean restructuring achieved by *imfs* overcomes structural bias present in the some of the benchmarks, which were likely derived from PLAs using a low-quality multi-level synthesis.

Finally, Section "Imfs+Lutpack" illustrates applying the proposed resynthesis (*lutpack*) to the results obtained by the other resynthesis (*imfs*). Given the power of *imfs* to overcome structural bias, it is somewhat unexpected that *lutpack* can achieve additional 5.4% of area reduction on top of *imfs*.

This additional area reduction speaks for the orthogonal nature of the two types of resynthesis. While *imfs* tries to reduce area by analyzing alternative resubstitutions at each node, it cannot efficiently compact large fanout-free cones that may be present in the mapping. The latter is efficiently done by *lutpack*, which iteratively collapses fanout-free cones up to 16 inputs and derives their new implementations using minimum number of LUTs.

The runtime of one run of *lutpack* did not exceed 20 sec for any of the benchmarks reported in Table 1. The total runtime of the experiments was dominated by *imfs* which has not yet been tuned for speed. Improved runtime measurements, with a break-down for different aspects of resynthesis, may be included in the final version of the paper.

8 Conclusions and future work

The paper presented a fast algorithm for matching Boolean functions with LUT structures of a fixed type and for mapping Boolean functions into the minimum number of K-input LUTs. The new algorithm is based on cofactoring and disjoint-support decomposition and is much faster than previous solutions that rely on BDD-based decomposition and Boolean satisfiability.

Area-oriented resynthesis of networks after LUT-mapping was investigated as one of the applications of the proposed Boolean matching algorithm. The resynthesis engine developed has given promising results after both good-quality technology mapping (area reduction 7.1%) and after another powerful resynthesis engine (area reduction 5.4%). This speaks for the usefulness of the proposed resynthesis, which collapses fanout-free cones and re-implements them using a minimal number of LUTs.

Future work in this area will include:

- Improving the DSD-based analysis, which is currently both the reason for occasional failure to find a feasible match and the most time-consuming part of matching.
- Exploring other data structures for cofactoring and DSD, to allow the algorithm to work for functions with more than 16 inputs. This will increase the size of functions processed by matching and improve the quality of resynthesis.

Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, and the California Micro Program with industrial sponsors Actel, Altera, Calypto, Magma, Synopsys, and Synplicity.

The authors are indebted to Stephen Jang of Xilinx for his masterful experimental evaluation of the proposed algorithms.

References

[1] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification", *Proc. DAC '05*, pp. 379-384.

[2] Actel Corp., "ProASIC3 flash family FPGAs datasheet," http://www.actel.com/documents/PA3_DS.pdf

[3] Altera Corp., "Stratix II device family data sheet", 2005, http://www.altera.com/literature/hb/stx/stratix_section_1_vol_1.pdf

[4] R. L. Ashenhurst, "The decomposition of switching functions", *Proc. Intl Symposium on the Theory of Switching*, Part I (Annals of the Computation Laboratory of Harvard University, Vol. XXIX), Harvard University Press, Cambridge, 1959, pp. 75-116.

[5] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70911. http://www.eecs.berkeley.edu/~alanmi/abc/

[6] V. Bertacco and M. Damiani. "The disjunctive decomposition of logic functions". *Proc. ICCAD '97*, pp. 78-82.

[7] D. Chai and A. Kuehlmann, "Building a better Boolean matcher and symmetry detector," *Proc. DATE '06*, pp. 1079-1084.

[8] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor cuts", *Proc. ICCAD '06*, pp. 143-150. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cut.pdf

[9] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*, pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[10] D. Chen and J. Cong. "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," *Proc. ICCAD '04*, 752-757.

[11] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA'99*, pp. 29-36.

[12] A. Curtis. *New approach to the design of switching circuits*. Van Nostrand, Princeton, NJ, 1962.

[13] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," *Proc. ASP-DAC '04*, pp. 591-596.

[14] A. Farrahi and M. Sarrafzadeh, "Complexity of lookup-table minimization problem for FPGA technology mapping," *IEEE TCAD*, Vol. 13(11), Nov. 1994, pp. 1319-1332.

[15] C. Files and M. Perkowski, "New multi-valued functional decomposition algorithms based on MDDs". *IEEE TCAD*, Vol. 19(9) , Sept. 2000, pp. 1081-1086.

[16] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping", *Proc. ICCAD '07*.

[17] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. Ph. D. Thesis. University of Michigan, 2001.

[18] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE TCAD*, Vol. 16(8), 1997, pp. 813-833.

[19] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: A study of optimality", *Proc. DAC '05*, pp. 427-432.

[20] V. Manohara-rajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *Proc. IWLS '04,* pp. 14-21.

[21] Y. Matsunaga. "An exact and efficient algorithm for disjunctive decomposition". *Proc. SASIMI '98*, pp. 44-50.
[22] K. Minkovich and J. Cong, "An improved SAT-based Boolean matching using implicants for LUT-based FPGAs," *Proc. FPGA'07*.
[23] A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis", *Proc. IWLS '02*, pp. 115-120. http://www.eecs.berkeley.edu/~alanmi/publications/2002/iwls02_enc.pdf
[24] A. Mishchenko, X. Wang, and T. Kam, "A new enhanced constructive decomposition and mapping algorithm", *Proc. DAC '03*, pp. 143-148.
[25] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC'06*, pp. 532-536.
[26] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cec.pdf
[27] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs". *IEEE TCAD*, Vol. 26(2), Feb 2007, pp. 240-253. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tcad06_map.pdf
[28] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis". Submitted to FPGA'08. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fpga08_imfs.pdf
[29] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_map.pdf
[30] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.
[31] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang. "Decomposition of multiple-valued relations". *Proc. ISMVL'97*, pp. 13-18.
[32] S. Plaza and V. Bertacco, "Boolean operations on decomposed functions", *Proc. IWLS' 05*, pp. 310-317.
[33] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient SAT-based Boolean matching for FPGA technology mapping," *Proc. DAC '06*.
[34] T. Sasao and M. Matsuura, "DECOMPOS: An integrated system for functional decomposition," *Proc. IWLS'98*, pp. 471-477.
[35] N. Vemuri and P. Kalla and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs", *ACM TODAES*, Vol. 7, 2002, pp. 501-525.
[36] B. Wurth, U. Schlichtmann, K. Eckl, and K. Antreich. "Functional multiple-output decomposition with application to technology mapping for lookup table-based FPGAs". *ACM Trans. Design Autom. Electr. Syst*. Vol. 4(3), 1999, pp. 313-350.

Table 1. Experimental evaluation of resynthesis applied to networks after technology mapping for FPGAs (K = 6).

Designs	PI	PO	Reg	Baseline		Choices		Lutpack		Imfs		Imfs + Lutpack	
				LUT	Level	LUT	Level	LUT	Level	LUT	Level	LUT	Level
alu4	14	8	0	821	6	785	5	580	5	558	5	453	5
apex2	39	3	0	992	6	866	6	788	6	806	6	787	6
apex4	9	19	0	838	5	853	5	790	5	800	5	732	5
bigkey	263	197	224	575	3	575	3	575	3	575	3	575	3
clma	383	82	33	3323	10	2715	9	2538	9	1277	8	1222	8
des	256	245	0	794	5	512	5	468	4	483	4	480	4
diffeq	64	39	377	659	7	632	7	635	7	636	7	634	7
dsip	229	197	224	687	3	685	2	685	2	685	2	685	2
ex1010	10	10	0	2847	6	2967	6	2552	6	1282	5	1059	5
ex5p	8	63	0	599	5	669	4	509	4	118	3	108	3
elliptic	131	114	1122	1773	10	1824	9	1827	9	1820	9	1819	9
frisc	20	116	886	1748	13	1671	12	1706	12	1692	12	1683	12
i10	257	224	0	589	9	560	8	555	8	548	7	547	7
pdc	16	40	0	2327	7	2500	6	2311	6	194	5	171	5
misex3	14	14	0	785	5	664	5	583	5	517	5	446	5
s38417	28	106	1636	2684	6	2674	6	2632	6	2621	6	2592	6
s38584	12	278	1452	2697	7	2647	6	2609	6	2620	6	2601	6
seq	41	35	0	931	5	756	5	712	5	682	5	645	5
spla	16	46	0	1913	6	1828	6	1594	6	289	4	263	4
tseng	52	122	385	647	7	649	6	649	6	645	6	645	6
geomean				1168	6.16	1103	5.66	1025	5.60	716	5.24	677	5.24
Ratio1				**1.000**	**1.000**	**0.945**	**0.919**	**0.878**	**0.909**	**0.613**	**0.852**	**0.580**	**0.852**
Ratio2						**1.000**	**1.000**	**0.929**	**0.989**	**0.649**	**0.926**	**0.614**	**0.926**
Ratio3										**1.000**	**1.000**	**0.946**	**1.000**