Combinational Equivalence Checking through Function Transformation

Hee Hwan Kwak, In-Ho Moon, James H. Kukula, and Thomas R. Shiple
Synopsys, Inc. USA.
{hkwak,mooni,kukula,shiple}@synopsys.com

ABSTRACT

Circuits can be simplified for combinational equivalence checking by transforming internal functions, while preserving their ranges. In this paper, we investigate how to effectively apply the idea to improve equivalence checking. We propose new heuristics to identify groups of nets in a cut, and elaborate detailed aspects of the new equivalence checking method. With a given miter, we identify a group of nets in a cut and transform the function of each net into a more compact representation with less variables. These new compact parametric representations preserve the range of nets as well as of the cut. This transformation significantly reduces the size of intermediate BDDs and enables the verification to be conclusive for many designs which state-of-the-art equivalence checkers fail to verify. Iterative groupings and transformations are performed until no grouping is possible for a cut. Then we proceed to the next cut and continue until the compare point is reached. Our experimental results show the effectiveness of our strategy and new grouping heuristics on the new method.

Categories and Subject Descriptors

J.6 [**Computer-aided engineering**]: Verification

General Terms

Algorithms, Experimentation, Verification

Keywords

Combinational verification, equivalence checking

1. INTRODUCTION

The process of checking the equivalence of combinational circuits has become a key component of verification flows in industry. In recent years, several approaches [5, 7, 8, 11, 15, 16] have been proposed to improve the performance of combinational equivalence checking. Despite the recent improvements, there are still many real world designs that are hard for state-of-the-art combinational equivalence tools to determine their equivalences.

There have been several attempts to improve combinational equivalence checking by reducing the size of intermediate BDDs [3]. The cut based algorithms [2, 8] were proposed to avoid building monolithic BDDs. They identify a cut and introduce new free variables to represent nets in a cut with the hope that the BDDs for the other internal functions to which nets in a cut fanout become smaller than those of the original functions. Even when they can successfully generate BDDs for compare points in designs under verification, they suffer from false negatives [2].

Due to false negatives, verification cannot tell if two designs are really different, and it is necessary to perform false negative resolution techniques. Even though multiple techniques have been developed to resolve the false negative problem, resolving false negatives still requires significant effort.

To overcome the false negative problem, Moondanos *et al.* [15] proposed the *normalized function* method. Instead of simply introducing a free variable for a net on a cut in the design, they replace a net with a simplified function containing a free variable, which is called a *normalized function*. Their limitation is that their *normalization* can only applied to a single net, whereas groups of multiple nets can be parameterized in our method.

On the other hand, Cerny *et al.* [5] proposed a technique based on range computation, which also does not suffer from false negatives. They compute the range of a cut from primary inputs and the reverse range from primary outputs. Then equivalence checking can be done by checking whether the reverse range covers the range. In this method, once a set of gates is composed to compute the range, the variables feeding only those gates are quantified, just as the fanout-free variables are quantified in the normalized function. However, this method suffers from BDD blowup since they compute the range of all nets in a cut, whereas our method computes one range at a time for a subset of nets and frees a computed range once it is parameterized. Furthermore, they require the success of range computation, whereas we can take another group if the range computation fails.

Based on the observations of previous work, our motivation becomes to reduce the size of intermediate BDDs by altering the functions of internal nets while maintaining the range of internal nets and in consequence, that of a miter [2], so that combinational equivalence checking can

$

proceed without BDD blowups and without false negatives.

In general, transformation of the functions of nets to more compact parametric representations [1, 6, 9, 14] with less variables can make the BDD size of new parametric representation smaller than that of the original nets. The reduction in the number of variables may alter the functions of nets since some original input variables are removed and new variables are introduced. However, since the original range of nets is preserved with the new parametric representations, our new method does not suffer from false negatives. The equivalence checking method we propose in this paper is based on range computation and parameterization.

In the remainder of this paper, We briefly review range computation and parameterization in Section 2. The conceptual overview of our method is given in Section 3. In Section 4, we focus on equivalence checking. Experimental results are shown in Section 5, and we analyze the test results in Section 6. We finally close in Section 7 with concluding remarks.

2. PRELIMINARY

We have presented a new theory to simplify circuits by preserving the range of all combinational outputs [14]. Even though we change the functions of the outputs, it is a valid transformation for verification of designs since we preserve the range. We here briefly review the theory.

Range computation and parametric representation are the main techniques that we have used in order to simplify circuits by altering the functions of the outputs while preserving the range of the outputs.

The key idea is that we first select a group containing a subset of outputs from all the outputs, then compute the range of the group by keeping only the set of variables that feed any unselected nets, then finally parameterize the computed range to get simplified functions of the group.

2.1 Definition of K and Q

Since we compute the range of a group instead of all the nets in a cut, we need to determine the variables to quantify in the range computation, and the ones to keep. This is because the simplicity of the parameterized functions depends on how many variables are quantified.

Consider a *cut* in a circuit under verification. Let C be the set of all the nets in the cut, V be the set of all support variables of C, N be the set of variables in V that are fanout-free that is, $\forall x \in N$, x feeds only one net in the cut, and R be the set of variables feeding more than one net. Now we formally define K and Q.

DEFINITION 2.1 (NON-BLOCKED). *A variable x is non-blocked iff there is more than one $f \in C$ such that $f^{\exists}$ or $f^{\forall}$ contains x, where $f^{\exists} = \exists_N.f$ and $f^{\forall} = \forall_N.f$.*

DEFINITION 2.2 (K). *K is a set of variables in V such that $K = \{x \mid x$ is non-blocked and $x \in R\}$.*

DEFINITION 2.3 (Q). $Q = V \backslash K$.

Now let G be a subset of C and L be the set of variables in K feeding only the nets in G. All the variables in L can be quantified in building the relation of G. Thus we can define the extended K and G for the group G.

DEFINITION 2.4 (K_G, Q_G). $K_G = K \backslash L, \quad Q_G = Q \cup L$.

2.2 K-set Preserving Range Computation

Let $\mathbb{B}$ be a Boolean domain and suppose we have a vector of Boolean functions $\vec{f} : \mathbb{B}^n \rightarrow \mathbb{B}^m$ with the input variables $\vec{x} = [x_1, \ldots, x_n]$ and output variables $\vec{y} = [y_1, \ldots, y_m]$.

Then the conventional range computation of $\vec{f}$ is defined by

$$R(\vec{y}) = \exists_{\vec{x}}. \bigwedge_{i=1}^{m} (y_i \equiv f_i(x_1, \ldots, x_n)). \quad (1)$$

Now we define a special type of range computation. Suppose $\vec{f}$ is the group G. The input variables $\vec{x}$ are split into K_G and Q_G as defined in Definition 2.4 and we quantify only Q_G so that we preserve the relations between the variables in K_G and the range. This is called *K-set preserving range computation* and is formally defined as

$$R(\vec{y}, K_G) = \exists_{Q_G}. \bigwedge_{i=1}^{m} (y_i \equiv f_i(K_G, Q_G)).$$

2.3 K-set Preserving Parametric Representation

Kukula and Shiple presented a method of parametric representation to generate circuits from the relation BDD of the design environment [10]. This method deals with the output variables of the environment as well as the input variables that depend on the states of the design under verification. Note that a parameterization used in [1] cannot be used for our purpose since the state dependent variables cannot be handled. We have extended Kukula's method to generate BDDs directly so that we do not need to build BDDs for the parameterized circuits.

Once the K-set preserving range of the group is computed, we try to get simpler functions of the group by parameterizing the range so that we preserve the range of the group as well as the range of the cut with the parameterized functions.

In this parameterization, the input variables are the ones in K and the output variables are the range variables. Thus we parameterize only the range variables by preserving the relations with the K variables. Each node of an output variable y in the relation BDD is parameterized as below.

$$\begin{aligned} y_p &= (\tilde{y}_{on} \wedge v) \vee \neg \tilde{y}_{off} \\ &= (\exists_Y. R_y \wedge v) \vee \neg \exists_Y. R_{\neg y}, \end{aligned}$$

where y_p is the parameterized sub-result of y on the node, $\tilde{y}_{on}$ is the quantified positive cofactor by quantifying all output variables Y, similarly $\tilde{y}_{off}$ is the quantified negative cofactor, and v is a parametric variable newly introduced for y.

Then the final parameterized results for y can be computed by collecting all y_ps of y by considering the care set of the node. For each output variable y_i below y in the relation BDD, the parameterized form of y_i can be obtained by merging two sub-results from the two children of the node as below.

$$\begin{aligned} (y_i)_p &= bdd\_ite(y_p, (y_i)_t, (y_i)_e) \\ &= (y_p \wedge (y_i)_t) \vee (\neg y_p \wedge (y_i)_e), \end{aligned}$$

where $(y_i)_t((y_i)_e)$ is the sub-result of y_i on the *then* (*else*) branch child node, respectively.

3. OVERVIEW

In this section we give an overview of our method with an example. Consider a miter shown in Figure 1. Given two functions $g_1 = abcde$ and $g_2 = d(a(b+ce) + \bar{a}\bar{c}e)$, there is no valuation of inputs a, b, c, d, and e that makes $g_1 = 1$ and $g_2 = 0$. Now consider two different functions $\hat{g_1} = v_1$ and $\hat{g_2} = v_1 + v_2$. Note that there is also no valuation of inputs v_1 and v_2 that makes $\hat{g_1} = 1$ and $\hat{g_2} = 0$. This implies that if we can come up with some new functions for g_1 and g_2, such as $\hat{g_1}$ and $\hat{g_2}$, we can reduce the number of input variables from 5 down to 2, while the range of g_1 and g_2 are preserved. Consequently in general, we can simplify the circuit and keep the BDDs smaller than the original functions and still preserve the range of the functions. Our intention is to reduce the size of intermediate BDDs for the internal nets in a miter while equivalence checking proceeds from inputs toward an output.

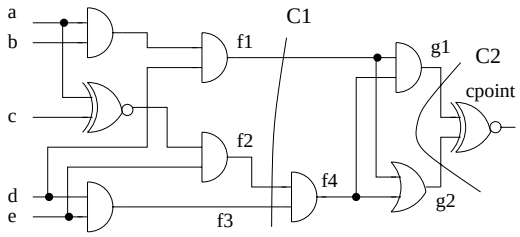

Figure 1: An Example Circuit

Now we illustrate conceptual steps of how to get new functions $\hat{g_1}$ and $\hat{g_2}$ from given functions g_1 and g_2 with ranges preserved. Consider a cut C_1. We first get N and R for C_1. Since b only fans out to f_1, and c only fans out to f_2, N becomes $\{b, c\}$ and R becomes $\{a, d, e\}$. From $f_1^{\exists} = ad$, $f_1^{\forall} = 0$, $f_2^{\exists} = 1$, $f_2^{\forall} = 0$, $f_3^{\exists} = de$, and $f_3^{\forall} = de$, we know that d and e are non-blocked, and K becomes $\{d, e\}$ and Q becomes $\{a, b, c\}$. Note that a is now in Q.

Let f_1 and f_3 be the group G_1. Since d is blocked by G_1 and does not fanout to any other nets in C_1, we can also move d from K to Q so that $K_{G_1} = \{e\}$ and $Q_{G_1} = \{a, b, c, d\}$. The *K-set* preserved range R_{G_1} of G_1 is the range of two function f_1 and f_3, which is

$$\begin{aligned} R_{G_1} &= \exists_{Q_{G_1}}.((f_1 \equiv abd)(f_3 \equiv de)) \\ &= f_3 e + \bar{f_3}(\bar{e} + \bar{f_1}) \end{aligned}$$

Note that f_1 and f_3 are output variables and e is an input variable. With the variable order $f_1 < f_3 < e$, we get $\hat{f_1} = v_1$ and $\hat{f_3} = e(v_1 + v_2)$, after parameterization. Now we replace f_1 and f_3 with $\hat{f_1}$ and $\hat{f_3}$, respectively. Note that the range of C_1 is preserved as well as that of f_1 and f_3 in the simplified circuit.

With the given new functions, we can group $f_2 = (ac + \bar{a}\bar{c})e$ and $\hat{f_3} = e(v_1 + v_2)$ to further simplify the circuit. Let f_2 and $\hat{f_3}$ be the group G_2. Then we have $K_{G_2} = \{v_1\}$ and $Q_{G_2} = \{a, c, e, v_2\}$. The range R_{G_2} is

$$\begin{aligned} R_{G_2} &= \exists_{Q_{G_2}}.((f_2 \equiv (ac + \bar{a}\bar{c})e)(\hat{f_3} \equiv e(v_1 + v_2))) \\ &= \hat{f_3} + \bar{v_1} + \bar{f_2} \end{aligned}$$

After parameterization with the variable order $f_2 < \hat{f_3} < v_1$, we have new functions for f_2 and $\hat{f_3}$ such that $\hat{f_2} = v_3$ and $\tilde{f_3} = v_1 v_3 + v_4$. The newly simplified circuit is shown in Figure 2. We stop here and proceed to the next cut.

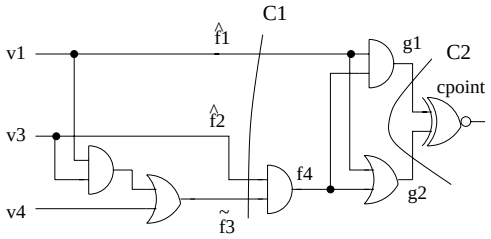

Figure 2: A Simplified Circuit

Now consider the cut C_2. Note that f_4 is now changed to $f_4 = \hat{f_2}\tilde{f_3} = v_3(v_1 + v_4)$, and g_1 becomes $g_1 = \hat{f_1} f_4 = v_1 v_3$, and g_2 becomes $g_2 = \hat{f_1} + f_4 = v_1 + v_3 v_4$. Once we move to the next cut C_2 and perform range computation and parameterization of g_1 and g_2 with the variable order $g_1 < g_2$, we get the range $\bar{g_1} + g_2$ and the parametric representation for g_1 and g_2 of $\hat{g_1} = v_5$ and $\hat{g_2} = v_5 + v_6$, respectively. The range of $\hat{g_1}$ and $\hat{g_2}$ is clearly the same as the original one. In this example, the verification fails since *cpoint* is not a constant one (i.e., $\bar{v_5} v_6$.)

4. EQUIVALENCE CHECKING

In this section, we give a detailed explanation of the overall flow of equivalence checking based on range computation and parameterization. We also elaborate the strategies that we have developed to improve the effectiveness of range computation and parameterization for equivalence checking. The importance of grouping is also mentioned in this section, and we propose heuristics that balance the performance and the quality of groups that are identified.

4.1 Overall Flow and Strategy

The overall flow of equivalence checking with *K-set* preserving range computation and parametric representation is shown in Figure 3. We first level a miter so that cuts can be easily recognized as explained in Section 4.2. For each cut, we generate BDDs for each net in the cut. If we cannot generate BDDs due to resource limitations, we simply abort the process. We check if a compare point is reached, and if so, we return the verification result. For each cut, we construct a support matrix SM, which is a matrix in which columns represent support variables and rows represent nets in a cut.

We proceed to group nets in a cut in order to simplify the circuit. SM is updated prior to the grouping, if it is modified by the previous parameterization. To compute K and Q, we first find the set N that contains all the variables that only feed one net in the cut. Similarly we identify a the set R. Initially we let K and Q be R and N, respectively. We apply existential quantification and universal quantification to each net to quantify out variables in N from the function of each net. That is, we compute $f^{\exists}$ and $f^{\forall}$, where for all f in a cut, $f^{\exists} = \exists_N.f$ and $f^{\forall} = \forall_N.f$. Then, for all $x \in K$ if x is a *non-blocked* variable, then it remains in K. Otherwise, it is moved from K to Q.

Note that these are very important procedures for the success of range computation since more variables are identified as Q and will be quantified out during range computation.

IdentifyGroup applies the heuristics discussed in Section 4.3, and returns an identified group G, which is a set of nets. With an updated support matrix, we compute K_G and Q_G

```
CheckEquivalence(Miter) {
  do {
    Set S contains all nets in the current cut
    Build BDDs for all nets in S
    if (failed to build BDDs) return ABORT
    if (reached the compare point)
      if (BDD_IS_ONE)
        return EQUIVALENT
      else
        return INEQUIVALENT
    Create Support Matrix SM for the current cut
    do {
      Compute K and Q
      if (G = IdentifyGroup(SM)) {
        Compute K_G and Q_G
        if (R_G = ComputeRange(G, Q_G))
          Parameterize(R_G, K_G)
        Update SM
      }
      else BREAK
    } while (TRUE)
    Advance to the Next Cut
  } while (TRUE)
}
```

Figure 3: Overall procedure for equivalence checking.

as follows. Let K_G and Q_G be K and Q initially, respectively. Since we now have a group G which blocks variables $\vec{g}$ that only fanout to nets in G, we can move variables $\vec{g}$ in K into Q_G, and remove $\vec{g}$ from K_G. Let the relation of G be T_G then the *K-set* preserved range is $R_G = \exists_{Q_G}.T_G$. A parametric representation for each net in G can be generated from R_G. Note that with new parametric representations for nets in G, we not only preserve the range of nets in G, we also preserve the range of the cut. We repeat this process until no group is identified in a cut. Then we advance to the next cut toward a compare point and repeat the same procedures until we reach a compare point.

Due to the simplification of function of each cut, range computation becomes much more efficient, and this gives us a better chance to get the parametric representations for the nets in the next cut. Note that our *K-set* preserved range computation is only needed for one group at a time. That is, we only perform one *K-set* preserving range computation for each group, and hence there exists only one range BDD for each group, which is freed once the parameterization is finished.

Even though we try to keep the cost of range computation as low as possible, there are still cases where it costs too much. To prevent the range computation from using too much resource, we abort the range computation when it exceeds our resource limit, as explained in Section 4.4, Furthermore, all its previous computations are nullified, and it tries to do new range computation for the next group.

In our experience, it is better to spend more time on finding good quality groups than just passing a bad group and performing range computation on it. Basically, we try to reduce the frequency of range computation aborts by identifying good groups. Section 4.3 gives the detailed explanation of how we find and determine good groups.

4.2 Cut Identification

We simply levels gates in a miter from inputs and set cuts based on those levels for our experiments. Since only nets in the same cut can be searched to make a group, how we set cut can be a major factor that affects the quality of groups for parameterization. One possible way to have better cuts would be using existing techniques for circuit partitioning such as multilevel min-cut partition algorithm [4]. Another possible way to get better cuts would be using random simulation to identify the equivalence classes of nets in a miter and set cuts accordingly.

4.3 Function Grouping

Since the range computation is an expensive operation, we need to simplify functions of nets in a cut as much and early as possible, so that the range computation has more chance to be performed successfully. Among many factors that affect the frequency of the success of range computation, we focus on group identification.

4.3.1 Group Quality Determination

Range computation is very expensive for some groups and is easy for others. Even though we abort the range computation if it exceeds our computation limits, we need to avoid the frequency of aborts in order to improve the performance of our method. There could be many ways to define the quality of the group. In our framework, we can say that the quality of a group is good if we can compute the *K-set* preserving range and perform parameterization with it. However, it is difficult to predict the quality of a given group prior to the range and parametric computations. Nonetheless, there are some properties that we can use to determine the quality of groups. They are as follows.

$$|Q| > |G| \tag{2}$$

$$|G| + |K| < T \tag{3}$$

Q and G are a *Q-set* and a group, respectively, and T is a threshold. Condition (2) makes sure that the number of removed variables should be greater than the newly introduced ones. Condition (3) is devised from the fact that the hardness of the range computation may depend on the size of K and G.

Both conditions (2) and (3) are used for both grouping heuristic 1 and 2 explained in Section *4.3.2*. There is one more condition we have devised for Heuristic 3, and it is explained in Section *4.3.2*.

4.3.2 Grouping Heuristics

Enumerating all the possible cases of grouping can be very costly and will not be feasible for most cases. In this section we propose new heuristics that balance the performance and the quality of groups that are formed. Our heuristic to find a group consists of three steps. We start from the most time efficient method and progress to heavier ones.

Heuristic 1. We find a group by looking at only one variable in the support matrix at a time. For each variable x in the support matrix, a list of nets $\vec{l}$ that x supports is collected. The intention is that at least variable x can be eliminated. Then for each $\vec{l}$, we compute its K set and Q set. With K and Q, we determine the quality of each $\vec{l}$ based on the condition mentioned in Section *4.3.1*. If $\vec{l}$ satisfies the condition, we put it into a candidate list. Among the candidate list, we usually select the smallest one, and it becomes a group.

Heuristic 2. If we cannot find a group from the previous step, we find a group by looking at two variables in a support matrix at a time, still, using the same quality measure as we did in the previous step. This step requires more computations than the previous one.

Heuristic 3. If we fail to find any group through the previous steps, we transform a support matrix using the MLP (Minimal Lifetime Permutation) technique [12] to a lower triangularized form, and group identification is attempted. As an example, we show a triangularized support matrix on the right in Figure 4, which is transformed from the one on the left.

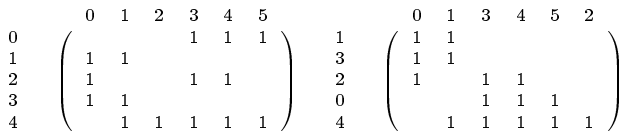

Figure 4: Lower Triangularized Support Matrix

With a given triangularized support matrix, more groups can be easily identified. Suppose we group nets in row $2, 0$, and 4. It is straight forward to recognize that variables in column $3, 4, 5$ and 2 will be quantified out. Since the groups found in this stage tend to be hard for range computation, we perform more comprehensive group quality checking procedures explained below.

Estimating Hardness of Range Computation. In Equation 3, we use K and G to estimate the hardness of the range computation since those variables will remain in the range. However we also want to consider Q to estimate the hardness. If there is no good quantification schedule for the variables in Q in the range computation, the number of variables in Q will also affect the hardness significantly.

Moon *et al.* proposed the use of variable lifetimes in the dependence matrix [13]. Also Moon *et al.* proposed the MLP (Minimal Lifetime Permutation) method to get a good quantification schedule by trying to decrease variable lifetime by using the dependence matrix [12].

We use the variable lifetime in the MLP method to see whether there is a good quantification schedule for Q. For this purpose, we first take an initial matrix to contain only the variables K_G and Q_G of the group G. Then we perform the MLP method and we compute the variable lifetime with respect to only Q since the variables in Q are quantified during the range computation. Finally if the variable lifetime is larger than a threshold, we discard the selected group since the variable lifetime implies that the range computation would be hard.

4.4 Resource Limiting in BDD operations

Range computation is already known as a very expensive operation and is very hard to guess whether a range computation is feasible within a given time and space.

Even with the estimations using Equations (3) and variable lifetime, range computations cannot be completed in a given time and space in many cases. To avoid this situation, we limit our time and space resources so that the range computation aborts once the resource limit is hit. However, the abort of one range computation does not mean that our overall algorithm aborts since we can select another group and continue.

We also apply this strategy to not only the range computation but also parameterization with different thresholds of the resource limits. It turns out that this resource limiting makes the overall algorithm work well without getting into cases where a BDD operation takes a long time.

5. EXPERIMENTAL RESULTS

The experiments were conducted on a Sun Fire Workstation running at 750Mhz with 4 CPUs and 16GB of RAM. In order to evaluate the effectiveness of the presented equivalence checking method, we carried out experiments using 12 test cases which are combinations of deep, shallow, and large or small single compare point miters that have gone through sequences of state-of-the-art solvers and have been aborted due to the complexity of their functions. In general, most of internal equivalent points are merged when those solvers are deployed. Hence, it is safe to assume that our test cases do not have internal equivalent points and constant nets in their cone.

The distribution of the circuit sizes and experimental results are shown in Table 1. Each test is a single compare point logic cone extracted from real world designs. $Test_F$ and $Test_G$ are obtained from the same design. The first three columns represent the number of cells in a cone (Cells), the total level of a cone (Level), and the number of primary inputs (PI), respectively. Column Time and Mem give the amount of verification time in seconds and the amount of peak memory consumed in MByte, respectively. In the Result column, a number appearing in parentheses right next to the word "abort" represents the level of a cone when it aborts. The last column (Vars) represents the number of remaining primary input variables and newly introduced parametric variables at the aborting cut level.

$Test_A$ through $Test_G$ are successfully verified and $Test_H$ through $Test_L$ are not. It is worth noting that $Test_F$ and $Test_G$ with 1284 initial primary inputs repeatedly reduce the number of their variables and solve their equivalences. For the aborted case, the rows of $Test_I$ and $Test_J$ indicate that there has been no change of the number of variables during verification. This implies no parameterization has been applied, and may suggest that there are no grouping possible on any cut, or our grouping method was not powerful enough to find ones. Note that our grouping algorithms presented in this paper do not search all the groups exhaustively, and if we have a different set of cuts, it would be possible for our current grouping algorithm to find groups. $Test_H$, $Test_K$, and $Test_L$ reduce the number of variables during verification and abort at cut 17, 26, and 27, respectively. These may be due to the fact that after the sequence of parameterizations the support matrix becomes full and no further grouping was possible.

Comparison with Normalization. From Table 1, it is easy to see that our method clearly outperforms the normalization method, which only succeeds in 3 out of 7 cases, where our method is successful. Even with the successful test cases (i.e., $Test_B$, $Test_C$, and $Test_E$), our method completes verification earlier than the normalization method on two test cases (i.e., $Test_C$ and $Test_E$). It is interesting to observe that the overall performance becomes better even though

Design	Cells	Level	PI	Our Method				Normalization [15]		
				Time	Mem	Result	Vars	Time	Mem	Result
$Test_A$	1203	78	131	4725	259	Equivalent		23355	943	Abort(35)
$Test_B$	3668	61	60	23366	401	Equivalent		9506	159	Equivalent
$Test_C$	2546	47	57	755	75	Equivalent		2760	52	Equivalent
$Test_D$	30804	89	303	13189	338	Equivalent		10274	252	Abort(16)
$Test_E$	29489	82	306	10205	298	Equivalent		16024	416	Equivalent
$Test_F$	5061	92	1284	1155	103	Equivalent		8628	940	Abort(14)
$Test_G$	5061	92	1284	1180	82	Equivalent		12709	964	Abort(14)
$Test_H$	22654	43	202	7572	979	Abort(17)	190	8160	995	Abort(17)
$Test_I$	1826	40	204	15978	516	Abort(24)	204	14741	555	Abort(24)
$Test_J$	3014	61	32	16479	1193	Abort(19)	32	15417	1222	Abort(19)
$Test_K$	8630	113	801	19674	1042	Abort(26)	317	77513	1002	Abort(26)
$Test_L$	1486	37	148	9681	665	Abort(27)	128	5218	82	Abort(25)

Table 1: Test Cases and Experimental Results

range computations and parameterizations are more expensive than normalization.

There is one test case $Test_B$ that our method is slower than normalization. This is due to the overhead of range computation failures. We were only able to compute 8 out of 52 ranges for $Test_B$ successfully. Since we did not explore all the possible groups, we may be able to improve the performance on $Test_B$ if we can find better groups.

6. ANALYSIS

In this section we perform further analysis on our experimental results.

6.1 Analysis on the Results with Parameterization

To highlight the effectiveness of multiple applications of grouping and parameterization on each cut, we present Figure 5 and Figure 6 which show the graphs with the number of variables and the number of groups identified at each cut, respectively. Even a single run of grouping, $K-set$ preserving range computation and parameterization finds a group of nets that blocks a subset of their supports, while multiple iterations of grouping substantially increase the chance to find more compact parametric representations with less variables for each net.

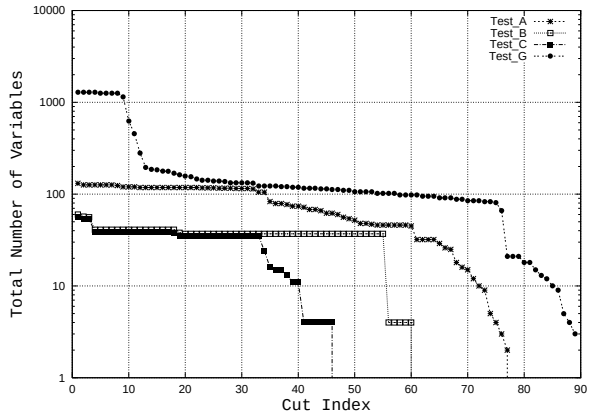

Figure 5: Total Number of Variables

The verification of $Test_G(\bullet)$ in Figure 5 starts initially with 1284 variables. At cut 4, 16 groups are identified, and the reduction of 27 variables occurs as a result. Between cut 8 and cut 14, over 1100 variables are reduced through more than 400 groupings decreasing the number of variables to 184. This implies that a substantial number of variables is locally blocked and is removed by the sequence of multiple iterations of grouping and parameterization. Furthermore, as shown in Figure 5, parameterization applied to nets close to a compare point effectively reduces the number of variables. This may be because most variables have a local affect and the internal nets close to a compare point may not be directly affected by many primary inputs.

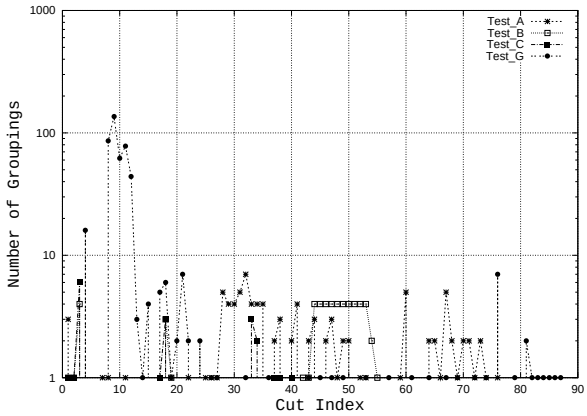

Figure 6: Number of Groups Identified

From Figure 6 it is interesting to observe that groups exist over a large number of cuts, even ones close to a compare point. Note that there are large groups that reduce only by a small amount the number of variables, whereas some small groups remove more variables. Even with hard miters such as our test cases, there exist many nets that can be grouped together, so that all the locally blocked variables in their supports can be removed.

Figure 7 shows the amount of peak memory that each test case consumes. Consider the line for $Test_G(\bullet)$. There is a rapid increase around cut 9, where a large number of parameterizations are applied, as shown in Figure 6. However, once the number of variables start to reduce from cut 10, the amount of peak memory is not increased any further. In general, during the repetitive computation of $K-set$ preserving range computation and parameterization, the amount of peak memory increases. This is due to the fact that the size of BDDs during range computation becomes large even though the BDD size of the resulting range could be small.

The overhead of $K-set$ preserving range computation varies on different test cases.

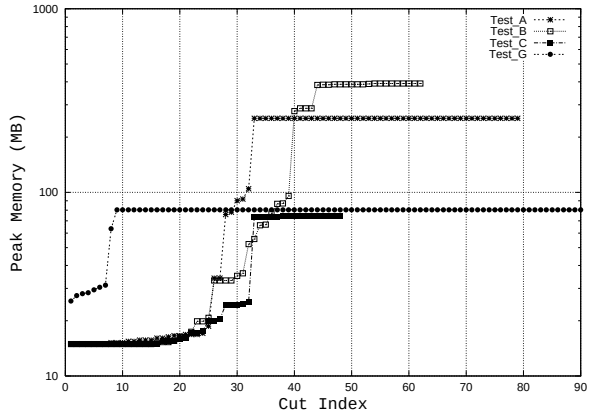

Figure 7: Peak Memory

To further illustrate the relation between the number of variables and peak memory consumed, a graph which plots the number of variables and the amount of peak memory for $Test_K$ is given in Figure 8, which aborts at cut 26. It reduces the 801 initial primary inputs down to 317 before it aborts. Peak memory increases steadily during the cut where grouping, range computation, and parameterization were possible, and from around cut 18 it grows rapidly where grouping was not possible any more. This is because around cut 18 the support matrix becomes full, so that every variable in the support matrix affects every net in the cut.

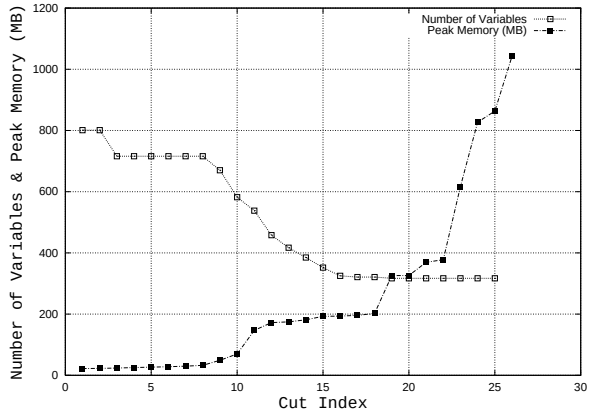

Figure 8: Number of Variables & Peak Memory of $Test_K$

6.2 Comparison to the Results without Parameterization

For comparison purposes, we disable our groupings, range computations, and parameterizations, and run all the test cases. Our experimental results show that all the tests abort. We list $Test_A$ through $Test_G$ in Table 2. As shown, there is no variable reduction.

To compare the amount of peak memory consumption of the two experiments, we present a graph showing the result of $Test_C$ and $Test_G$ in Figure 9. Consider the line ($\odot$) of

	Vars	Time(Secs)	Mem(MB)	Result
$Test_A$	131	7507	298	Abort(43)
$Test_B$	60	52691	854	Abort(32)
$Test_C$	57	23948	942	Abort(29)
$Test_D$	303	9693	247	Abort(16)
$Test_E$	306	4359	196	Abort(24)
$Test_F$	1284	9729	476	Abort(15)
$Test_G$	1284	14491	997	Abort(16)

Table 2: Test Cases without Parameterization

$Test_G$ with our method and a line ($\bullet$) of $Test_G$ without our method. Clearly, $\bullet$ grows exponentially after cut 14 and hits the memory limit, whereas $\odot$ does not consume more than 822 MBytes.

During cuts 1 through 14, $\odot$ consumes more memory than $\bullet$ due to the repetitive range computations and parameterizations. From cut 1 through 7, range computations were performed 16 times. Hence, the gap between $\odot$ and $\bullet$ mainly comes from 16 range computations. As shown, this overhead of $K-set$ preserving range computation for multiple groups is minimal.

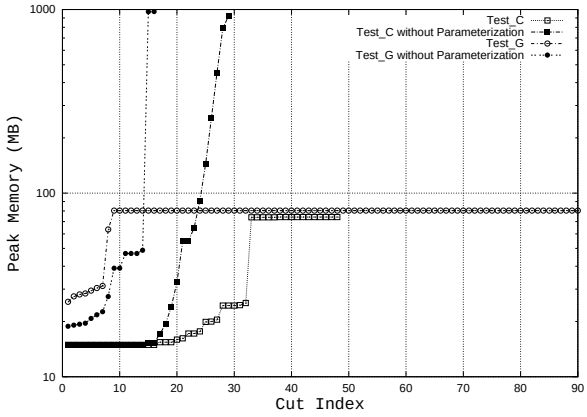

Figure 9: Peak Memory with VS. without Parameterization

At cuts 8 and 9, range computations were performed 86 and 136 times, respectively. The gap between $\odot$ and $\bullet$ during these two cuts is clearly larger than that of the previous cuts, but not by far. The two lines $\odot$ and $\bullet$ show that even with the potential overhead of computing $K-set$ preserving ranges, our method consumes significantly less memory by reducing the size of intermediate BDDs. This is due to the fact that the cost of parameterization is very cheap, and that the parametric representation is very compact so that the size of BDDs becomes small. For $Test_C$, the first 14 cuts consume about the same memory. This is because few small groups were found.

We show Figure 10 to illustrate that the reduction of the number of variables is a powerful way to reduce the intermediate memory consumption during equivalence checking. It clearly shows that without our method, all the initial primary inputs remain to affect the internal functions until it aborts.

7. CONCLUSIONS

As the experiments in Section 5 show, equivalence check-

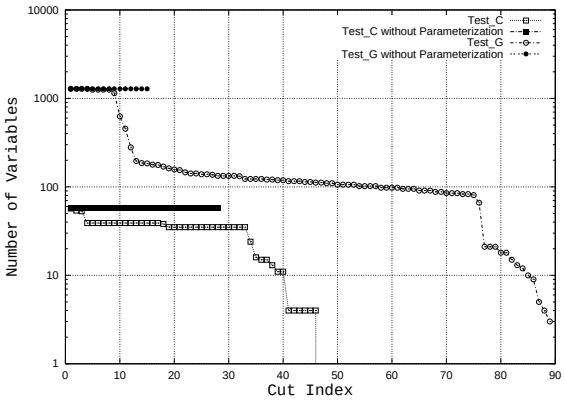

Figure 10: Number of Variables with VS. without Parameterization

ing based on multiple range computations and parameterizations is very promising in producing robust verification results.

The grouping phase is able to find groups of internal nets which block their supports effectively. Iterative multiple applications of the grouping and parameterization have been selectively performed by estimating the feasibility and the gain from the computation. Furthermore, once the range is computed, we do not keep the range. Instead we get a parametric representation, which is usually more compact.

Finding good quality groups is also an important factor that reduces the frequency of range computation aborts. Further research may identify better schemes to identify cuts so that grouping algorithms, range computation, and parameterization work even better. Approaches to partition the circuit based on structure may permit finding better ways to define cuts. New powerful variant of grouping strategy can be also developed with further research.

In this paper, we have shown that our framework of range computation and parametric representation is ideally suited to equivalence checking.

8. REFERENCES

[1] M. Aagaard, R. B. Jones, and C.-S. H. Seger. Formal verification using parametric representations of boolean constraints. In *Proceedings of the Design Automation Conference*, pages 402–407, June 1999.

[2] D. Brand. Verification of large synthesized designs. In *Proceedings of the International Conference on Computer-Aided D esign*, pages 534–537, Santa Clara, CA, Nov. 1993.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[4] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved algorithms for hypergrph bisection. In *Proceedings of ASP-DAC*, 2000.

[5] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. In *Proceedings of the International Conference on Computer-Aided D esign*, pages 34–37, Santa Clara, CA, Nov. 1990.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, Nov. 1989.

[7] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proceedings of Design, Automation and Test in Europe*, pages 114–121, 2001.

[8] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the Design Automation Conference*, pages 263–268, Anaheim, CA, June 1997.

[9] J. H. Kukula and T. R. Shiple. Building circuits from relations. In E. A. Emerson and A. P. Sistla, editors, *12th Conference on Computer Aided Verification (CAV'00)*, pages 131–143. Springer-Verlag, Chicago, July 2000. LNCS 1855.

[10] J. H. Kukula and T. R. Shiple. Building Circuits from Relations. In *Proceedings of Computer Aided Verification Conference (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[11] J. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of Design, Automation and Test in Europe*, pages 145–149, 1999.

[12] I.-H. Moon, G. D. Hachtel, and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 73–90. Springer-Verlag, Nov. 2000. LNCS 1954.

[13] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the Design Automation Conference*, pages 23–28, Los Angeles, CA, June 2000.

[14] I.-H. Moon, H. H. Kwak, J. H. Kukula, T. R. Shiple, and C. Pixley. Simplifying circuits for formal verification using parametric representation. In *Formal Methods in Computer Aided Design*, Portland, OR, November 2002.

[15] J. Moondanos, C.-S. H. Seger, Z. Hanna, and D. Kaiss. Clever: Divide and conquer combinational logic equivalence verification with false negative elimination. In B. Berry, H. Comon, and A. Finkel, editors, *13th Conference on Computer Aided Verification (CAV'01)*, pages 131–143. Springer-Verlag, Paris, July 2001. LNCS 2101.

[16] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Proceedings of Computer Design*, pages 459–464, 2000.