# Modular Partitioning and Dynamic Conjunction Scheduling in Image Computation

**Christoph Meinel** **Christian Stangier**
**FB Informatik, University of Trier**

## Abstract

Image computation is the core task in any formal verification applications like reachable states computation or model checking. In OBDD-based image computation a partitioned representation of the transition relation is used. The quality of the partitioning and the schedule in which the partitions are processed is crucial for the efficiency of the image computation. In this paper we describe an approach to build a hierarchical modular partitioned transition relation. Based on this partitioning we present a dynamic conjunction scheduling algorithm that improves the flexibility and computational power of the image computation. The concept is proven by symbolic model checking experiments.

## 1 Introduction

The computation of the reachable states of a finite state machine (FSM) is an important task for synthesis, logic optimization and formal verification. The increasing complexity of sequential systems like controllers or protocols requires efficient reachable states computation methods. If the reachable states are computed by using Ordered Binary Decision Diagrams (OBDDs) [2], the system under consideration is represented in terms of a transition relation. Since the monolithic representation of the circuit's transition relation usually leads to unmanageable large OBDD-sizes, the transition relation has to be partitioned [3, 5]. The quality of the partitioning is crucial for the efficiency of the reachable states computation.

The partitioning problem consists of two parts: First, the latches of the FSM have to be clustered. Second, the clusters have to be scheduled for the image computation.

Many approaches [8, 15, 10] perform an ordering of the latches before clustering. Instead we follow the paradigm of grouping strongly related latches and perform clustering only within those groups. In [14] an approach was presented that derives a hierarchical partitioning from RTL information of the design. We show how to obtain a hierarchical modular partitioning without using external information. The benefit of the hierarchical partitioning is that it provides a good basis for dynamic conjunction scheduling. The dynamic scheduling algorithm presented here is targeted to optimize the AndExist operation, which is underlying the image computation.

The rest of the paper is structured as follows: In the next section we describe the concept of a partitioned transition relation and the image computation using such a transition relation, also an overview of recent work on this area is given. The next two sections present our new technique for building a modular transition relation and the algorithm for dynamic conjunction scheduling. In Section 5 we give an experimental proof of the concept. The last section draws conclusions.

## 2 Preliminaries

### 2.1 Partitioned Transition Relations

The computation of the reachable states is a core task for optimization and verification of sequential systems. The essential part of OBDD-based traversal techniques is the transition relation:

$$\mathrm{TR}(x, y, e) = \prod_i \delta_i(x, e) \equiv y_i,$$

which is the conjunction of the transition relations of all latches ($\delta_i$ denotes the transition function of the $i$th latch, $x, y, e$ represent present state, next state and input variables). This *monolithic* transition relation is represented as a single OBDD and usually is much too large to allow computation of the reachable states. Sometimes a monolithic transition relation is even too large for a representation with OBDDs. Therefore, more sophisticated reachable states computation methods make use of a *partitioned* transition relation [3], i.e. a cluster of OBDDs each of them representing the transition relation of a group of latches. A transition relation partitioned over sets of latches $P_1, \ldots, P_j$ can be described as follows:

$$\mathrm{TR}(x, y, e) = \prod_j \mathrm{TR}_j(x, y, e) \text{ , where}$$

$$\mathrm{TR}_j(x, y, e) = \prod_{i \in P_j} \delta_i(x, e) \equiv y_i.$$

### 2.2 Image Computation using AndExist

The reachable states computation consists of repeated image computations $Img(\mathrm{TR}, R)$ of a set of already reached states $R$:

$$Img(\mathrm{TR}, R) = \exists_{x,e}(\mathrm{TR}(x, y, e) \cdot R)$$

With the use of a partitioned transition relation the image computation can be iterated over $P_i$ and the $\exists$ operation can be applied during the product computation *(early quantification)*:

$$Img(\mathrm{TR}, R) = \exists_{v^j}(\mathrm{TR}_j \cdot \ldots \cdot \exists_{v^2}(\mathrm{TR}_2 \cdot \exists_{v^1}(\mathrm{TR}_1 \cdot R) \ldots),$$

where $v^i$ are those variables in $(x \cup e)$ that do not appear in the following $\mathrm{TR}_k$, $(i < k \leq j)$.

The so called *AndExist* [3] or *AndAbstract* operation performs the AND operation on two functions (here partitions) while simultaneously applying existential quantification ($\exists_{x_i} f = (f_{x_i=1} \vee f_{x_i=0})$) on a given set of variables, i.e. the variables that are not in the support of the remaining partitions. Unlike the conventional AND operation the AndExist operation only has a exponential upper bound for the size of the resulting OBDD, but for many practical applications it prevents a blow-up of OBDD-size during the image computation.

Since the number of quantified variables depends on the order in which the partitions are processed, finding an optimal order of the partitions for the AndExist operation is an important problem. We refer to this problem as the *conjunction scheduling problem*. Geist and Beer [8] presented a heuristic for scheduling of partitions each representing a single state variable. The so called *IWLS95-method* [15] computes a conjunction schedule by using a greedy scheme to minimize the number of variables involved in the AndExist operation.

**Related Work**

After the IWLS95 method has been the standard method for partitioning for several years, recently new approaches have been published:

Moon, Hachtel and Somenzi [10] presented a heuristic that minimizes *active lifetime* of the variables in the product to gain a good conjunction schedule. The active lifetime is the number of conjunctions in which the variable is involved schedule and it is computed from a dependency matrix, which describes the dependency between the different latches. Additionally, the authors give a blocking strategy for the clustering, which forbids clustering across certain borders that have turned out to produce too large clusters.

Meinel and Stangier [12] presented a method to utilize RTL-information from the design to group related variables. A dependency matrix is used in [13] to find groups of strongly related latches. In [14] a hierarchical partitioning based on structural information from the design and a strategy for dynamic conjunction scheduling is presented.

Chauhan et.al. [6] presented a heuristic that creates a parse tree resulting in a non-linear quantification schedule. The heuristic argument is to perform the cheapest (in OBDD-size) conjunctions first. This strategy also allows dynamic conjunction scheduling.

Cabodi, Camurati and Quer [7] presented an improved benefit heuristic, which allows adjustment of the partitioning when reordering occurs. Also, they give an adaptive clustering approach, allowing dynamic conjunction scheduling and clustering.

Gupta et.al. [9] introduced a hybrid method that combines OBDD and SAT techniques to reduce the complexity of the OBDD-operations. Another hybrid method by Moon et.al [11] combines transition functions and relations.

Except for [14] and [6], the basic idea of the pure OBDD approaches still is ordering of bit-relations. This work will focus on the grouping aspect of latches that has been proven to be a powerful concept.

# 3 Building a Modular Transition Relation

Before images can be computed the transition relation has to be built. It is possible to rebuild the transition relation dynamically for any image computation, but this is very costly. Instead, the partitioned transition relation should be flexible enough to adapt to changing requirements.

The paradigm we follow to build a partitioned transition relation is to group semantically related latches. In [14] it was shown how a hierarchical partitioning supports an efficient image computation and allows dynamic conjunction scheduling. Here, we want to build a hierarchical partition of the transition relation by exploiting the structure of the matrix representing the latch to latch dependencies.

In the following we present our algorithm for modular grouping of the latches. Than, we describe the algorithm for building OBDD clusters in these groups and discuss the effects.

## 3.1 The Grouping Algorithm

Our algorithm follows a *separate and group* strategy. Therefore, we utilize a *latch dependency matrix* (LDM). An entry of the LDM gives the number of variables that latch $l_i$ and latch $l_j$ have in common, i.e.

$$LDM(l_i, l_j) = |supp(l_i) \cap supp(l_j)|.$$

A higher number denotes a high dependency and thus a strong interaction of the latches. A low number reflects a weak relation of the latches.

The grouping algorithm proceeds in three phases, where in the first two phases *separated* modules of latches are build, while in the third phase *grouping* inside the modules takes place. The phases in detail:

**1. Module definition phase:** During this phase we create completely independent modules $M_i$. In this phase the modules contain only a single latch, which serves as a representative for the module. A latch will be a representative for a new module, whenever there is no dependency with the representatives of the other modules: $\forall_j : LDM(l_i, M_j) = 0$. This operation is performed for all latches. The threshold for the dependency is 0 to keep a reasonably small number of modules. The latches are checked consecutively. The latches that do not form an own module go in the set $R$ of the remaining latches. The modules $M_i$ and the remaining latches $R$ serve as a basis for the 2nd phase.

**2. Module assignment phase:** The remaining latches of $R$ are assigned to the modules using a best-fit strategy, i.e. a latch $l_i$ is put into a module $M_k$ containing the latch $l_j$, which has the highest common dependency with latch $l_i$:

$$l_i \rightarrow M_k : \quad LDM(l_i, l_j) = max, \quad l_j \in M_k.$$

A minimum dependency is required, otherwise this latch is a representative for a new module. At the end of this phase all latches are assigned to modules. The value of the minimum dependency controls the number of additional modules.

**3. In-module grouping phase:** Now, we have medium to strongly related latches inside the modules. To build groups within the modules we have to apply a more complex heuristic than those before. The heuristic has to be able to group latches that are strongly related, but also has to avoid building one large group containing all latches. Please keep in mind: All latches in a single module have a certain dependency at least to one other latch in the module.

We propose a grouping algorithm based on merging of groups that utilizes a *group dependency matrix* (GDM). An entry of the GDM denotes the number of shared variables from the *common support* of group $G_i$ and group $G_j$. The common support of a group is defined as the intersection of the support of all members of the group:

$$GDM(G_i, G_j) = |supp(G_i) \cap supp(G_j)|.$$

The algorithm works as follows for a module $M$:

1. Initialize the groups, such that each group contains a single latch $G_i = l_i, l_i \in M$ and compute the initial GDM.
2. Compute the maximum dependency in the GDM $maxdep = \max_{i,j}(GDM(G_i, G_j))$.
3. Pairwise merge groups $G_i$ and $G_j$ whose dependency equals *maxdep*.
4. Update the GDM, the support of a merged group is the intersection of the support of the former groups: $supp(G_i) = supp(G_i) \cap supp(G_j)$.
5. Repeat 2-4 until *maxdep* is below a certain threshold or the number of allowed runs is exceeded.

Step 2 of this phase guarantees that strongly related latches are grouped. Performing an intersection of the support of the groups that are merged (Step 4) and limiting the number of runs avoids construction of groups with latches that are related only very loosely.

The latches that could not be grouped after termination of the algorithm are considered being the modules own latches.

The runtime of this algorithm is cubic in the number of latches, but it is negligible in comparison to other operations during construction of the transition relation (OBDD-AND, variable reordering, etc.).

In our implementation we set the minimal required dependency for the 2nd phase to 3 variables. In phase 3 we set the threshold for maxdep to 5 and limited the number of runs to 10% of the number of latches.

### 3.2 Clustering

After the relations have been grouped, the clusters of the partitioned transition relation can be computed. A cluster of the transition relation is solely built from latches of one group. If the OBDD-size of a cluster exceeds a certain threshold an additional cluster for this group is created. The OBDD-size is a rather artificial indicator for the separation of clusters, but it is used in all partitioning approaches to avoid size explosion of the transition relation. Fortunately, the grouping approach drastically limits the influence of this threshold on the efficiency of the image computation by providing more meaningful borders for the clusters.

Figure 1 shows a schematic of the result of clustering the hierarchical partitioned transition relation. Starting from the root node (*Main*) that represents the whole design one reaches the *modules* (M1, M2, etc.) of the design. These modules hardly interact and can be seen as almost independent FSMs. Attached to the main node we find relations that hardly interact with any of the modules, but do not form an own module. Those relations may be seen as global state variables.

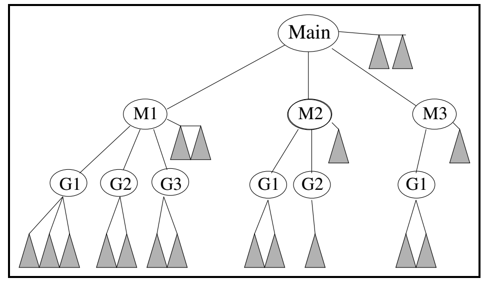

Figure 1: Result of the Modular Clustering.

Each module has a small number of children the so called *groups* (G1, G2, etc.). Relations within a group strongly interact and should stay as close as possible, in the best case within one cluster. Attached to each module we find those relations that do not interact strong enough to any of the groups to be attached to one of them. Keeping those loosely coupled relations separate in the module gives us more freedom in the choice of the conjunction schedule.

Summarized, the modular grouping approach results in a hierarchical partitioning consisting of a small number of separate modules each consisting of a reasonable number of groups whose relations strongly interact. The depth of the hierarchy is always limited to three layers, because the grouping algorithm only allows the creation of one main node, one module layer and one group layer. The algorithm is not recursive.

For computation of the image based on this hierarchical partitioning the algorithm for *hierarchical image computation* presented in [14] is used.

## 4 Dynamic Conjunction Scheduling

After the transition relation has been built, the conjunction scheduling problem has to be solved. The hierarchical partitioning does not imply a schedule for the conjunctions of the clusters, i.e. the order in which to apply the AndExist. It gives a basic idea of how to conjunct members of a single group, but there is no hint about how to schedule modules.

The following algorithm based on the heuristic in [14] exploits the concept of compact clusters to gain a schedule for the conjunctions that not only tries to optimize single AndExist operations, but is able to improve the entire image computation process, as the following discussion will show.

### 4.1 The Scheduling Algorithm

The algorithm is called recursively for all modules starting with the main module. When entering a new module a decision is made whether to start the conjunction operations with the submodules (children) of this module or its own cluster. The decision is made depending on the maximum level of a variable to be quantified out (for the submodules this value has to be computed in advance). The group with the smaller value is computed first. Notice that for any chosen cluster these values will change as the sets of variables to be quantified out change, too. The group with the smallest value is chosen first. So, BFS- and DFS-style recursion may interleave during the recursion, but not within a module. Within a module the children resp. the modules own clusters are ordered by the same argument and chosen by increasing values.

After each AndExist step the quantify variables have to be recomputed. Whenever there is a change in the variable order of the OBDD, the ranking of the modules and clusters is changing too, thus resulting in a completely dynamic conjunction schedule.

## 4.2 Discussion of the Algorithm

The efficiency of the dynamic conjunction scheduling algorithm described in the previous section relies on the *compact clustering* produced by the hierarchical partitioning. Compact clustering means that there are fewer support variables and a higher number of quantify variables in each cluster. As a side effect we can expect that variables of a compact cluster will stay close in the variable order due to their high interaction even during dynamic variable reordering.

To understand the effect of compact clustering on the AndExist operation let us take a closer look at the algorithm (see Figure 2). The AndExist operation performs the quantification of variables ($\exists_{x_i} f = (f_{x_i=1} \vee f_{x_i=0})$)) while performing a recursive AND. Thus, it follows the general scheme of any recursive OBDD algorithm like ITE:

- First, terminal cases and the computed table are checked (lines 1, 2).
- If the actual variable (*top*) is not to be quantified, the recursion is called recursively with the successors of the current nodes (10, 11) and a node is created from the results $t, e$ of the recursion.
- If no more variables are to be quantified out, the recursion switches to the regular AND (3, 8).
- If the actual variable is to be quantified out, the recursion continues with the successors of $f$ and $g$ (14, 15) as well, but afterwards, an OR operation on the results $t, e$ of the recursion is executed (16).

Let us recall: The OR operation is called within the recursive step of the AndExist! This actually is the reason why AndExist is not a polynomial operation. To make things even worse the results $t$ and $e$ of the recursive step are obsolete later and are dereferenced (17), i.e. if not part of another OBDD they will be deleted. This means the AndExist operation produces – in contrast to any other OBDD operation – temporary nodes. These temporary nodes are the main reason for a possible blow-up in OBDD-size during image computation.

```
    AndExistRecur(f,g,quanvars){
1)    Check_terminal_cases;
2)    /*sink-nodes, recomputation, etc.*/
3)    if(!quanvars) return BddAnd(f, g);
4)    if(g == ONE) return BddExist(f, quanvars);
5)    top = topmost variable in f and g;
6)    top_q = topvar in quanvars;
7)    while(top_q < top) top_q = top_q→T;
8)    if(top_q == ONE) return BddAnd(f, g);

9)    if(top_q > top){ /* no quantify */
10)       t = AndExistRecur(f→T, g→T, quanvars);
11)       e = AndExistRecur(f→E, g→E, quanvars);
12)       return makeBDDnode(index of top, t,e);
13)   }else{ /* quantify */
14)       t = AndExistRecur(f→T, g→T, quanvars→T);
15)       e = AndExistRecur(f→E, g→E, quanvars→T);
16)       result = BddOr(t, e);
17)       Deref(t); Deref(e);
18)       return result;
      }
    }
```

Figure 2: Sketch of the AndExist Algorithm.

We think that the problem of temporary nodes is more profound than the problem of having the OR operation inside the recursion. We try to reach shortcuts (3, 4, 8) in the computation earlier, to keep the lifetime of temporary nodes shorter. For this reason we will even accept a slightly more complex OR operation.

In a compact clustering as shown (simplified) in Figure 3a we have fewer variables in a cluster ($T_i$) of the transition relation than in a regular clustering as shown in Figure 3b. If the clusters are ordered by choosing the cluster with the highest maximum level of a quantify variable first, we can expect earlier shortcuts in the AndExist operation. As the clusters have fewer variables, the AND operation (3, 8) terminates earlier (e.g. AND$(f, 1) = f$) leaving the bottom part of $R$ untouched.

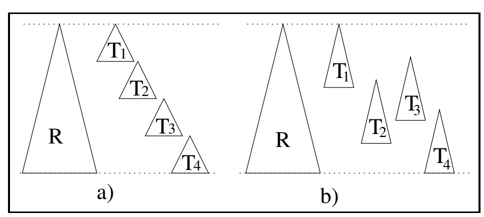


Figure 3: Schematic of Compact Clustering (a) vs. Regular Clustering (b).

We can conclude that the dynamic conjunction scheduling does not only try to improve single AndExist operations (e.g. by performing the cheapest operations in terms of OBDD-size first), instead the whole series of AndExist operations will be optimized. As the output of the AndExist operation is (except for the last cluster) always an input to the next AndExist operation, this heuristic optimizes the complete image computation process.

## 5 Experiments

We implemented our algorithms in the VIS-package [4] (version 1.3) using the underlying CUDD-package [16] (version 2.3.0). VIS is a popular verification and synthesis package in academic research. It inherits state of the art techniques for OBDD manipulation, image and reachable states computation as well as formal verification techniques.

### 5.1 Benchmarks

For our experiments we used Verilog designs from the Texas97 benchmark suite [1] and from the VIS distribution. This publicly available benchmark suite contains real life designs from industry and academics including: MSI Cache Coherence Protocol, PCI Local BUS, PI BUS Protocol, MESI Cache Coherence Protocol, MPEG System Decoder, DLX and PowerPC 60x Bus Interface. The benchmark suite also contains properties given in CTL formulae for verification.

Only those designs were considered, whose transition relation could be build respecting our system limitations. We computed 25 different benchmarks for which one or two sets of properties have been checked (resulting in 41 experiments). The runtime heavily depends on the chosen set of properties to be checked and is not proportional to the number of image computations. Therefore, it is reasonable to check more than one set of properties. Some very small examples (CPU time $<$ 5s) are not shown.

### 5.2 Experimental Setup

We left all parameters of VIS and CUDD unchanged. The most important default values are: Partition cluster size = 5000, Partition method for MDDs = frontier, Dynamic OBDD variable reordering method = sifting, First reordering threshold = 4004 nodes. Before building the transition relation the OBDD was minimized by an explicit call to variable reordering. The CPU time was limited to 6 CPU hours and memory usage was limited to 200MB. All experiments were performed on Linux PentiumIII 500Mhz workstations.

### 5.3 Results

We compare our method (Groupmod) to the MLP method of [10]. For results on runtime and space requirements see Table 1. Icmp is the sum of forward and backward image computations performed during the analysis. Parts gives the number of partitions of the transition relation. The OBDD-size of the transition relation cluster and the peak number of live nodes is given by TRn resp. Peakn. The CPU time is measured in seconds and given as Time. The columns denoted with % describe the improvement in percent[1].

Table 1 gives a detailed comparison of the results for the MLP method and the Groupmod method with dynamic conjunction scheduling. The Groupmod method wins in 34 of the 41 cases, resulting in an overall cputime improvement of 39%. In some – especially larger – cases the Groupmod method drastically improves OBDD-size and runtime (up to 87%). The Groupmod method performs worse only on seven small and mid-sized benchmarks.

Although the Groupmod method introduces more clusters (23%) than the MLP method, the overall number of OBDD-nodes for the transition relation is 12% smaller. This is a good indicator for the improved quality of the computed partitioning Another indicator is the overall peak node size that improved almost as good as the runtime (38%).

[1] $0 <$ improvement $< 100$; $-100 <$ impairment $< 0$.

## 6 Conclusion

We presented an approach for partitioning the transition relation that follows the idea of grouping strongly related latches. The resulting partitioning is flexible enough to allow a dynamic conjunction scheduling. The dynamic scheduling then works towards a more efficient AndExist operation. We believe that the modular group partitioning is so powerful that it might serve as a basis for more sophisticated dynamic scheduling algorithms than the one presented here.

The MLP method seems to work more efficiently with fewer clusters. A hybrid method combining MLP and Groupmod depending on the number of clusters and the quality of the CDM could be even more powerful.

## References


[1] A. Aziz et. al., *Texas-97 benchmarks*,http://www-cad. EECS.Berkeley.EDU/Respep/Research/Vis/texas-97.

[2] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 1986.

[3] J. R. Burch, E. M. Clarke and D. E. Long, *Symbolic Model Checking with partitioned transition relations*, Proc. of Int. Conf. on VLSI, 1991.

[4] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy and T. Villa, *VIS: A System for Verification and Synthesis*, Proc. of Computer Aided Verification, 1996.

[5] O. Coudert, C. Berthet and J. C. Madre, *Verification of Synchronous Machines using Symbolic Execution*, Proc. of Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, Springer, 1989.

[6] P. Chauhan, E. M. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith and D. Wang, *Non-linear Quantification Scheduling in Image Computation*, Proc. of Int. Conf. on CAD, 2001.

[7] G. Cabodi, P. Camurati and S. Quer, *Dynamic Scheduling and Clustering in Symbolic Image Computation* Proc. of Design and Test in Europe, 2002.

[8] D. Geist and I. Beer, *Efficient Model Checking by Automated Ordering of Transition Relation Partitions*, Proc. of Computer Aided Verification, 1994.

| | | MLP | | | | GROUPMOD | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Icmp | Peakn | Parts | TRn | Time | Peakn | % | Parts | TRn | % | Time | % |
| ONE.PPCliveness | 27 | 22522 | 4 | 2560 | 8 | 19246 | 14 | 6 | 1809 | 29 | 7 | 17 |
| ONE.contention | 18 | 22468 | 4 | 2541 | 8 | 19246 | 14 | 6 | 1809 | 29 | 6 | 23 |
| ONE.pixley_cpu | 89 | 22522 | 4 | 2560 | 10 | 26790 | -15 | 6 | 1809 | 29 | 12 | -20 |
| PCIabnorm.PCI | 200 | 114493 | 9 | 16560 | 206 | 114041 | 0 | 11 | 17787 | -6 | 237 | -12 |
| PCInorm.PCI | 118 | 85804 | 8 | 14989 | 112 | 69291 | 19 | 10 | 16749 | -9 | 62 | 45 |
| TWO.PPCliveness | 37 | 103691 | 7 | 8415 | 45 | 272317 | -61 | 11 | 7393 | 12 | 237 | -80 |
| TWO.contention | 34 | 117457 | 6 | 9128 | 40 | 139081 | -14 | 10 | 7909 | 13 | 85 | -52 |
| TWO.pixley_cpu | 81 | 109767 | 8 | 11370 | 64 | 465323 | -75 | 11 | 7393 | 35 | 403 | -83 |
| coherence | 79 | 16350 | 2 | 2526 | 5 | 12196 | 25 | 5 | 1512 | 40 | 5 | 15 |
| elevator | 30 | 25398 | 3 | 3411 | 10 | 16495 | 35 | 7 | 1528 | 55 | 6 | 42 |
| ethernet.define.213 | 236 | 2599643 | 8 | 4852 | 4349 | 275055 | 89 | 11 | 2989 | 38 | 572 | 87 |
| gcd | 26 | 215222 | 3 | 7941 | 48 | 215222 | 0 | 3 | 5707 | 28 | 44 | 9 |
| minMax30 | 6 | 101042 | 3 | 8073 | 15 | 101042 | 0 | 4 | 4521 | 44 | 13 | 19 |
| multi_main.multi | 44 | 33423 | 3 | 5598 | 64 | 33423 | 0 | 10 | 1842 | 67 | 19 | 71 |
| p62_LS_LS_V02.ccp | 53 | 130716 | 19 | 26351 | 130 | 61976 | 53 | 23 | 26699 | 0 | 63 | 52 |
| p62_LS_LS_V02.p6l | 75 | 130716 | 19 | 26351 | 133 | 65395 | 50 | 23 | 26699 | 0 | 71 | 47 |
| p62_LS_L_V02.ccp | 53 | 127627 | 18 | 33499 | 122 | 61810 | 52 | 23 | 25727 | 23 | 60 | 51 |
| p62_LS_L_V02.p6li | 70 | 127677 | 18 | 33495 | 124 | 63839 | 50 | 23 | 25727 | 23 | 65 | 47 |
| p62_LS_S_V02.ccp | 53 | 127627 | 18 | 33499 | 122 | 61810 | 52 | 23 | 25727 | 23 | 62 | 49 |
| p62_LS_S_V02.p6li | 70 | 127677 | 18 | 33495 | 124 | 63839 | 50 | 23 | 25727 | 23 | 66 | 47 |
| p62_L_L_V02.ccp | 52 | 148560 | 19 | 25128 | 176 | 79094 | 47 | 23 | 27616 | -8 | 63 | 64 |
| p62_L_L_V02.p6liv | 74 | 145689 | 19 | 26081 | 147 | 83610 | 43 | 23 | 27616 | -5 | 92 | 37 |
| p62_L_S_V02.ccp | 54 | 120656 | 19 | 30562 | 115 | 60657 | 50 | 23 | 25522 | 16 | 56 | 52 |
| p62_L_S_V02.p6liv | 75 | 120656 | 19 | 30562 | 116 | 60657 | 50 | 23 | 25522 | 16 | 57 | 51 |
| p62_ND_LS_V02.ccp | 102 | 226684 | 19 | 29424 | 407 | 134258 | 41 | 24 | 27155 | 8 | 234 | 42 |
| p62_ND_LS_V02.p6l | 144 | 687049 | 19 | 30483 | 1093 | 400977 | 42 | 24 | 27155 | 11 | 808 | 26 |
| p62_ND_L_V02.ccp | 138 | 2509642 | 21 | 31457 | 10640 | 1490002 | 41 | 24 | 28029 | 11 | 3240 | 70 |
| p62_ND_L_V02.p6li | 152 | 611021 | 21 | 31457 | 1434 | 555394 | 9 | 24 | 28029 | 11 | 1041 | 27 |
| p62_ND_S_V02.ccp | 83 | 199660 | 19 | 32197 | 329 | 96385 | 52 | 23 | 26308 | 18 | 130 | 60 |
| p62_ND_S_V02.p6li | 126 | 186164 | 20 | 33989 | 412 | 209625 | -10 | 23 | 26308 | 23 | 329 | 20 |
| p62_S_S_V02.ccp | 36 | 121454 | 18 | 26007 | 121 | 54447 | 55 | 23 | 27068 | -3 | 43 | 65 |
| p62_S_S_V02.p6liv | 55 | 122998 | 18 | 25939 | 121 | 54447 | 56 | 23 | 27068 | -3 | 63 | 48 |
| p62_V_LS_V02.ccp | 89 | 140414 | 19 | 28637 | 210 | 107902 | 23 | 24 | 29339 | -1 | 147 | 30 |
| p62_V_LS_V02.p6li | 131 | 178936 | 19 | 28637 | 301 | 510964 | -64 | 24 | 29339 | -1 | 1051 | -70 |
| p62_V_S_V02.ccp | 83 | 220119 | 20 | 26807 | 223 | 78180 | 64 | 23 | 25504 | 5 | 82 | 63 |
| p62_V_S_V02.p6liv | 126 | 220119 | 20 | 26807 | 239 | 144977 | 34 | 23 | 25504 | 5 | 222 | 7 |
| packet | 65325 | 56282 | 3 | 6139 | 11755 | 168874 | -66 | 4 | 8131 | -23 | 10540 | 10 |
| single_main.single | 81 | 18672 | 2 | 4301 | 22 | 9360 | 50 | 6 | 1831 | 57 | 7 | 67 |
| single_main.single1 | 34 | 18672 | 2 | 4301 | 22 | 9360 | 50 | 6 | 1831 | 57 | 7 | 68 |
| two_processor | 223 | 110358 | 3 | 5625 | 86 | 179562 | -37 | 7 | 1928 | 66 | 170 | -48 |
| two_processor_bin | 129 | 192179 | 3 | 6983 | 118 | 65156 | 66 | 6 | 2307 | 67 | 35 | 70 |
| Total: | | 10717826 | 494 | 778728 | 33824 | 6711325 | | 641 | 686166 | | 20509 | |
| Improvement: | | | | | | **38%** | | **-23%** | **12%** | | **39%** | |

Table 1: Comparison of MLP Method and Group Hierarchy Heuristic


[9] A. Gupta, Z. Yang, P. N. Ashar, L. Zhang and S. Malik, *Partition-Based Decision Heuristics for Image Computation Using SAT and BDDs*, Proc. of Int. Conf. on CAD, 2001.

[10] I. Moon, G. D. Hachtel and F. Somenzi , *Border-Block Triangular Form and Conjunction Schedule in Image Computation*, Proc. of Formal Methods in CAD, LNCS 1954, 2000.

[11] I. Moon, J. Kukula, K. Ravi and F. Somenzi, *To Split or to Conjoin: The Question in Image Computation*, Proc. of Design Automation Conf., 2000.

[12] Ch. Meinel and C. Stangier, *Speeding Up Image Computation by using RTL Information*, Proc. of Formal Methods in CAD, LNCS 1954, 2000.

[13] Ch. Meinel and C. Stangier, *A New Partitioning Scheme for Improvement of Image Compuation*, Proc. of ASP Design Automation Conf., 2001.

[14] Ch. Meinel and C. Stangier, *Hierarchical Image Computation with Dynamic Conjunction Scheduling*, Proc. of IEEE Int. Conf. on Computer Design, 2001.

[15] R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley and B. Plessier, *Efficient BDD Algorithms for Synthesizing and Verifying Finite State Machines*, Proc. of Int. Workshop on Logic Synthesis), 1995.

[16] F. Somenzi, *CUDD: CU Decision Diagram Package*, ftp://vlsi.colorado.edu/pub/ .