# On Solving Covering Problems

Olivier Coudert

Synopsys Inc., 700 East Middlefield Rd.
Mountain View, CA 94043, USA

## Abstract

The set covering problem and the minimum cost assignment problem (respectively known as unate and binate covering problem) arise throughout the logic synthesis flow. This paper investigates the complexity and approximation ratio of two lower bound computation algorithms from both a theoretical and practical point of view. It also presents a new pruning technique that takes advantage of the partitioning.

## 1 Introduction

The (unate or binate) covering problem is a well known intractable problem. It has several important applications in logic synthesis, such as two-level logic minimization, two-level Boolean relation minimization, three-level NAND implementation, state minimization, exact encoding, and DAG covering [10, 6, 2].

Although from the practical point of view people are more interested in low cost heuristic algorithms that produce approximate solutions, solving covering problems exactly is the only way to evaluate the performance of these heuristic algorithms. Moreover, improving exact solvers can help finding better heuristics.

Since a branch-and-bound based covering problem solver has to look at a search space that is potentially exponential, one has to prevent the solver from exploring unsuccessful branches. This relies on two aspects: lower bound computation and pruning techniques. We have introduced better lower bound computation techniques [4] as well as original pruning methods [5] that dramatically improved the efficiency of covering problem solvers.

|  | $\mathbf{y}_1$ | $\mathbf{y}_2$ | $\mathbf{y}_3$ | $\mathbf{y}_4$ |
|---|---|---|---|---|
| $\mathbf{y}_1 + \mathbf{y}_3 + \mathbf{y}_4$ | 1 |  | 1 | 1 |
| $\overline{\mathbf{y}_1} + \mathbf{y}_2 + \overline{\mathbf{y}_4}$ | 0 | 1 |  | 0 |
| $\mathbf{y}_2 + \overline{\mathbf{y}_3} + \mathbf{y}_4$ |  | 1 | 0 | 1 |
| $\overline{\mathbf{y}_2} + \mathbf{y}_3 + \overline{\mathbf{y}_4}$ |  | 0 | 1 | 0 |

Figure 1: A binate covering matrix.

This paper presents an extension of this work. Section 2 reviews some concepts related to binate covering problems. Section 3 addresses the lower bound computation problem. It discusses the quality of achievable lower bounds from a theoretical point of view. It shows that an original lower bound computation method has a better approximation ratio than the widely used independent set based method. Section 4 introduces a new pruning technique that takes advantage of the partitioning. Section 5 presents experimental evidences that demonstrate the effectiveness of these methods on unate and binate covering problems.

## 2 Binate Covering Problem

This section defines the binate covering problem and the notations that will be used in the sequel.

We denote $v$ an element of $\{0,1\}$, and $\mathbf{y}$ a Boolean variable. We denote $\mathbf{y}^v$ the assignment of $\mathbf{y}$ to $v$. Let $f(\mathbf{y}_1, \ldots, \mathbf{y}_n)$ be a Boolean function from $\{0,1\}^n$ into $\{0,1\}$. Let *Cost* be a function that associates a positive cost with the assignment of variable $\mathbf{y}_k$ to 0 or 1. The cost of a $n$-tuple $(v_1, \ldots, v_n)$ of $\{0,1\}^n$ is defined as $\sum_{k=1}^{n} Cost(\mathbf{y}_k^{v_k})$.

**Definition 1** *The binate covering problem consists of finding a minimum cost $n$-tuple that evaluates $f$ to 1.*

Let $\prod_{i=1}^{m} s_i$ be a product-of-sums representation of $f(\mathbf{y}_1, \ldots, \mathbf{y}_n)$. The covering matrix associated with a binate covering problem is a matrix $M$ made of $m$ rows

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | |
| $x_2$ | 1 | | | | | 1 | 1 | 1 | 1 | | | | | | |
| $x_3$ | | 1 | | | | 1 | | | | 1 | 1 | 1 | | | |
| $x_4$ | | | 1 | | | | 1 | | | 1 | | | 1 | 1 | |
| $x_5$ | | | | 1 | | | | 1 | | | 1 | | 1 | | 1 |
| $x_6$ | | | | | 1 | | | | 1 | | | 1 | | 1 | 1 |

Figure 2: An ill-conditioned covering matrix.

labeled with the sums $s_i$, and of $n$ columns labeled with the variables $\mathbf{y}_j$. An element $M[i,j]$ of the matrix is equal to 1 if $(\mathbf{y}_j \Rightarrow s_i)$, to 0 if $(\overline{\mathbf{y}_j} \Rightarrow s_i)$, and to 2 otherwise[1]. A $n$-tuple $(v_1, \ldots, v_n) \in \{0,1\}^n$ covers a row $s_i$ iff there exists a $j$ such that $M[i,j] = v_j$. The binate covering problem consists of finding a minimum cost $n$-tuple that covers all the rows of the matrix. Fig. 1 shows a covering matrix[2] for the function $\mathbf{y}_2\mathbf{y}_3 + (\mathbf{y}_1 \oplus \mathbf{y}_4)$. Assuming the same non zero cost for positive literals, and a zero cost for negative literals, the minimum cost true assignments are the 4-tuples $(1,0,0,0)$ and $(0,0,0,1)$.

We denote $\langle X, Y, \mathit{Cost}\rangle$ a covering problem, where $X$ and $Y$ are the sets of rows and columns respectively. We identify $\mathbf{y}^v$ with the set of rows $\mathbf{y}$ covers when it is assigned to $v$. We identify a row $x$ with the set of columns that cover it. The symbol $y$ will be used to denote a set $\mathbf{y}^v$, and we consider that $\mathbf{y}$ represents the pair of columns $(\mathbf{y}^0, \mathbf{y}^1)$. One says that $\mathbf{y}$ is unate if $\mathbf{y}^v = \emptyset$ for some $v$. A covering problem whose columns are all unate is a unate covering problem, also called a set covering problem.

A covering matrix can be simplified using essentiality, dominance [11, 9], partitioning, and Gimpel's reduction [7, 12]. The reader is referred to [2, 13, 5] for a review of these reduction techniques. Iterating the reduction process produces a fixpoint, called the cyclic core of the covering matrix [11]. If it is empty, a minimum solution is built from the essential columns found during the reduction, and from the columns used by Gimpel's reductions.

If the reduced covering problem $C = \langle X, Y, \mathit{Cost}\rangle$ is not empty, a branch-and-bound resolution can be performed. A branching variable $\mathbf{y}$ is selected to generate two covering problems. The first one, $C_v = \langle X - \mathbf{y}^v, Y - \{\mathbf{y}\}, \mathit{Cost}\rangle$, considers that $\mathbf{y} = v$, and the other, defined as $C_{\overline{v}} = \langle X - \mathbf{y}^{\overline{v}}, Y - \{\mathbf{y}\}, \mathit{Cost}\rangle$, considers that $\mathbf{y} = \overline{v}$. Both problems $C_v$ and $C_{\overline{v}}$ are recursively solved, which produces the minimum solution of $C$.

[1]Assuming there is no sum that contains a variable and its negation.

[2]We omit the 2's when showing covering matrix.

# 3 Lower Bound Computation

When solving a covering problem with a branch-and-bound algorithm, it is essential to compute a good lower bound of the problem generated at some branch of the search tree, since the branch can be pruned as soon as the lower bound is greater or equal to the best solution found so far. This section presents two lower bound computation methods, and shows that, from an approximation ratio point of view, the second method is far more effective than the widely used independent set based one. It then explains why in practice the latter is a better choice for most of the problems related to logic synthesis, although the second method is especially effective in some other applications.

## 3.1 Independent Set Based Lower Bound

A technique widely used to compute a lower bound consists of finding a set $L$ of one-to-one disjoint rows, i.e., there is no column covering more than one row of $L$. Consider the graph whose vertices are the rows, and such that there is an edge between two rows iff they are covered by a common column. Then $L$ is nothing but an *independent set* of this graph. The minimum cost needed to cover $L$ is

$$\begin{aligned} \mathit{Cost}(L) &= \sum_{x \in L} \mathit{Weight}(x), \qquad \text{where} \\ \mathit{Weight}(x) &= \min_{\substack{y \in Y \\ y \ni x}} \mathit{Cost}(y), \end{aligned}$$

which is a lower bound on the minimum cost needed to cover $X$.

Indeed, the independent set based lower bound can be arbitrarily far from the minimum cost solution of the binate covering problem. Consider a covering matrix made of $n$ rows and $\binom{n}{2}$ columns, such that any pair of rows is 1-covered by a single column, and each column covers only two rows[3]. Fig. 2 shows such a matrix for $n = 6$. Assuming a unit cost for all columns, the maximum independent set based lower bound is 1, while the minimum cost covering is made of $\lceil n/2 \rceil$ columns.

## 3.2 Approximation of Independent Set

Maximizing the independent set based lower bound is NP-hard. A natural heuristic computation consists of computing an independent set $L$ in a greedy way, as shown in Fig. 3. It consists of selecting a row $x$ of $X$ to be put in $L$, removing from $X$ all the rows that have a column in common with $x$ to enforce $L$ to be an independent set, and iterating this process until $X$ is empty.

[3]This polynomial size unate matrix is completely reduced w.r.t. to dominance relations and Gimpel's reduction for $n > 2$.

```
L ← Ø;
while  X ≠ Ø  do  {
      x ← an element of  X;
      L ← L ∪ {x};
      X ← X − ⋃_{y∋x} y;
}
return  L;
```

Figure 3: Greedy independent set computation.

We are now interested in comparing the best achievable independent set based lower bound with the one obtained using a greedy algorithm. The effectiveness of such a greedy algorithm depends on the choice of the row $x$ to be put in the independent set $L$. Let us assume that we select the row $x$ that maximizes the ratio of its weight (the greater, the better) and of its number of elements (the smaller, the better). Theorem 1 shows that it yields a $O(|X|)$ ratio approximation of the maximum independent set based lower bound, which is a very poor approximation[4]. The proof still holds for more complex selection heuristics, e.g., the one proposed in [4]. Although it has not been proved yet for all polynomial-time selection heuristics, it is unlikely that such a greedy approach can achieve a better approximation.

**Theorem 1** *Let $C = \langle X, Y, Cost \rangle$ be a binate covering problem. Let $L^o$ be a maximum cost independent set, and $L$ the independent set computed by the algorithm given in Fig. 3, where the heuristic selection is:*

$$x \quad = \quad \arg\max_{x \in X} \frac{Weight(x)}{|x|}$$

*Then we have:*

$$Cost(L) \leq Cost(L^o) \leq Cost(L) \max_{x \in X} |x|$$

*Proof.* Let $L = \{x_1, x_2, \ldots\}$ where $x_k$ is the $k$-th selected row. Since $L$ is a set of disjoint sets of columns, we can properly define

$$w(y) \quad = \quad \begin{cases} \frac{Weight(x_k)}{|x_k|} & \text{if } y \in x_k \\ 0 & \text{otherwise} \end{cases}$$

Let us show that $\sum_{y \in x} w(y) \geq \frac{Weight(x)}{|x|}$ for any $x$.

[4]The best known approximation ratio is $O(|X|/(\log |X|)^2)$, but the corresponding algorithm is more costly [1].

Note that since the independent set $L$ is maximal (i.e., any row intersects some row of $L$), there exists a least integer, say $j$, for which $x \cap x_j \neq \emptyset$.

$$\begin{aligned} \sum_{y \in x} w(y) &= \sum_k \sum_{y \in x \cap x_k} \frac{Weight(x_k)}{|x_k|} \\ &\geq \frac{Weight(x_j)}{|x_j|} \\ &\geq \frac{Weight(x)}{|x|} \end{aligned}$$

The last inequality holds because, by definition of $j$, we have $x \subseteq (Y - \cup_{k=1}^{j-1} x_k)$, and $x_j$ has been chosen because it maximizes the quantity given above.

Thus we have (the second inequality holds because $L^o$ is made of disjoint sets):

$$\begin{aligned} \sum_{y \in Y} w(y) &= \sum_k \sum_{y \in x_k} w(y) = \sum_k Weight(x_k) = Cost(L) \\ &\geq \sum_{x \in L^o} \sum_{y \in x} w(y) \\ &\geq \sum_{x \in L^o} \frac{Weight(x)}{|x|} \\ &\geq \frac{Cost(L^o)}{\max_{x \in X} |x|} \end{aligned}$$

□

### 3.3 log-Approximation of Unate Problems

We have shown that not only the maximum (NP-hard) independent set based lower bound can be arbitrarily far from the minimum cost solution, but also that its approximation with a greedy algorithm is of poor quality. We present an original lower computation technique that has the same complexity than the method presented above, but that guarantees a log approximation ratio w.r.t to the minimum solution on unate problem. This result still holds on binate covering problem when the method finds a feasible solution.

Indeed, this result relies on an inequality relating the cost of the best solution and the cost of a solution obtained in a greedy way. In other words, we first generate a feasible solution, and we then derive from its cost a lower bound on the minimum solution.

Let $\gamma$ be some strictly positive weighting function defined on the set of rows, and let us define

$$\Gamma(y) \quad = \quad \sum_{x \ni y} \gamma(x).$$

Fig. 4 presents a subquadratic time algorithm that build a solution[5] in a greedy way. From the cost of the solution

[5]Note that it is not necessary irredundant.

```
S ← Ø;
while  X ≠ Ø  do  {
      y  ← arg min_{y∈Y} Cost(y)/Γ(y∩X);
      X ← X − y;
      S  ← S ∪ {y};
}
return  S;
```

Figure 4: Greedy computation of a solution.

$S$ obtained by this algorithm, we derive a lower bound on the problem, as stated by the following theorem (a weaker form of this result can be found in [8, 3]).

**Theorem 2** *Let $C = \langle X, Y, Cost\rangle$ be a unate covering problem. Let $S^o$ a minimum cost solution of $C$, and $S$ the solution computed by the algorithm given Fig. 4. Then we have:*

$$\frac{Cost(S)}{r} \leq Cost(S^o) \leq Cost(S)$$

*where*

$$r = \sum_{k=1}^{\max_{y\in Y}\Gamma(y)} 1/k$$

*if $\gamma$ is integer valued, and*

$$r = 1 + \log \max_{y\in Y} \frac{\Gamma(y)}{\min_{x\in y}\gamma(x)}$$

*otherwise. Note that the later upper bounds the former.*

*Proof.* Let $S = \{y_1, y_2, \ldots\}$, where $y_k$ is the $k$-th selected column. Let us denote $S(y, k) = y - \cup_{j=1}^{k-1} y_j$ the set of $x$'s belonging to some $y$ that remains uncovered by the first $k-1$ selected columns. Thus $S(y_k, k)$ is the set of $x$'s that are newly covered by $y_k$ at the $k$-th step. Note that the sets $S(y_k, k)$ form a partition of $X$, so we can properly define for $x \in S(y_k, k)$:

$$w(x) = Cost(y_k)\frac{\gamma(x)}{\Gamma(S(y_k, k))}$$

We now show that $\sum_{x\in y} w(x) \leq r(y)Cost(y)$ for any $y$.

$$\sum_{x\in y} w(x) = \sum_k \sum_{x\in y_k\cap S(y,k)} Cost(y_k)\frac{\gamma(x)}{\Gamma(S(y_k, k))}$$

By definition, the sequence $S(y, k)$ is decreasing and eventually becomes empty for indices $k$ greater than some integer $j$. Also note that at the $k$-th step of the greedy algorithm, $y\cap X$ is nothing but $S(y, k)$. Since $y_k$ minimizes $\lambda y.\frac{Cost(y)}{\Gamma(S(y,k))}$, we obtain:

$$\begin{aligned}
\sum_{x\in y} w(x) &\leq Cost(y)\sum_{k=1}^{j}\frac{\Gamma(y_k \cap S(y, k))}{\Gamma(S(y, k))} \\
&= Cost(y)\sum_{k=1}^{j}\frac{a_k - a_{k+1}}{a_k},
\end{aligned}$$

where $a_k = \Gamma(S(y, k))$, since $y_k \cap S(y, k) = S(y, k) - S(y, k + 1)$.

If $\gamma$ is integer valued, then $(a_k)$ is a decreasing positive integer sequence that becomes zero at $k = j + 1$, so we have:

$$\begin{aligned}
\sum_{k=1}^{j}\frac{a_k - a_{k+1}}{a_k} &\leq \sum_{k=1}^{j}\sum_{i=a_{k+1}+1}^{a_k} 1/i \\
&= \sum_{i=1}^{a_1} 1/i \\
&= \sum_{i=1}^{\Gamma(y)} 1/i
\end{aligned}$$

In the general case, $(a_k)$ is a decreasing positive sequence that becomes zero at $k = j + 1$, and we have:

$$\begin{aligned}
\sum_{k=1}^{j}\frac{a_k - a_{k+1}}{a_k} &\leq 1 + \sum_{k=1}^{j-1}\int_{a_{k+1}}^{a_k}\frac{dx}{x} \\
&= 1 + \log\frac{a_1}{a_j} \\
&\leq 1 + \log \max_{y\in Y}\frac{\Gamma(y)}{\min_{x\in y}\gamma(x)}
\end{aligned}$$

We then have (the second inequality holds for any cover of $X$):

$$\begin{aligned}
\sum_{x\in X} w(x) &= \sum_k \sum_{x\in S(y_k,k)} w(x) = \sum_k Cost(y_k) = Cost(S) \\
&\leq \sum_{y\in S^o}\sum_{x\in y} w(x) \\
&\leq \sum_{y\in S^o} r(y)\,Cost(y) \\
&\leq r \cdot Cost(S^o)
\end{aligned}$$

□

Theorem 2 shows that unate covering problem is $\log(|X|)$-approximable (take $\gamma(x) = 1$). Although binate

covering problem is not polynomially approximable[6], the log ratio still holds when the greedy algorithm of Fig. 4 produces a feasible solution.

This log inequality holds for any $\gamma$, in particular for the branching column selection heuristics of [13] and [4]. Here, we are interested in minimizing $r$ and maximizing $Cost(S)$. Since $\gamma(x)$ captures the difficulty of covering $x$ (the greater, the more difficult), we can *reverse* the criterion that would yield a good upper bound in order to obtain a good lower bound (e.g., we can take $\gamma(x) = |x|$).

### 3.4 Discussion

Let us call *IND* the independent set based lower bound presented in Section 3.2, and *LB* the lower bound computation method presented in Section 3.3. We have shown that not only *IND* has a poor approximation ratio regarding the maximum independent set based lower bound, but also that the latter can be arbitrarily far from the minimum solution. On the other hand, *LB* has a log-ratio approximation of the minimum solution.

Intuitively, the greater the density of the covering matrix, the more edges between the rows in the induced graph, the smaller the maximum independent set. Thus we can expect low quality lower bound with *IND* on such problems, while *LB*, thanks to its log-approximation ratio, can be still effective. On some high density ($> 30\%$) optimal assignment problems in operations research, *LB* is typically 20% to 60% better than *IND*, and up to 5 times better.

Unfortunately, *LB*, although far better than *IND* from a theoretical point of view, yields a smaller lower bound than *IND* when the problem becomes smaller or sparser. Moreover, *IND* can take advantage of the *limit lower bound* [5], which can prune a complete search tree that *LB* could not prevent from being explored. In general, *LB* does not improve the lower bound enough to significantly increase the number of prunings, except for some very particular examples, e.g., ill-conditioned or high density covering matrices.

## 4 A Partition Based Pruning Technique

When the rows and columns of a covering matrix can be permuted so that its 0 or 1 elements only occur in a partition made of disjoint diagonal blocks $B_k$ as shown in Fig. 5, each block can be solved independently and concatenating their solutions yields a solution of the original problem [13]. This section describes a pruning method[7] that takes advantage of such a partitioning. We denote *C.lower* and *C.upper* some lower and upper bound of a set covering problem $C$. As soon as $C.lower \geq C.upper$, the branch $C$ belongs to can be pruned. When $C$ is solved, both *C.lower* and *C.upper* are set to the minimum cost solution. We call *block* a subproblem yielded by a partition.

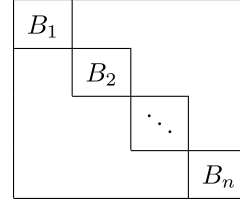


Figure 5: Partition of a covering problem

Fig. 6 shows how a partition is solved. The method relies on three aspects:

- Lower bounds of the blocks are computed to prune the resolution as soon as possible.
- Upper bounds of the blocks are computed and used with the lower bounds to overconstrain the resolution of blocks, thus reducing the space search the solver has to look at.
- Each block and its most recent resolution status is cached, so that it can be retrieved when another partitioning yields the same block.

The resolution status is as follows: *UNSAT* (the problem is not feasible), *SOLVED* (the minimum solution is known), *PRUNED* (one did not find a better solution than the current upper bound), *LU* (we only know a lower and upper bound[8] on the problem), and *NEW* (a new block with no information). When a partition of a problem $C$ is found, we call the function of Fig. 6 on the list *partition* of blocks. If a block is found in the cache, we retrieve its most accurate informations regarding its minimum solution, else we compute a lower and upper bound (which can solve the block if both these bounds are equal). If the resolution of $C$ is not pruned thanks to the lower bounds, we end up with a list *unsol* of unsolved blocks. Each of these blocks is then solved, but its space search is restricted by forcing a possibly overconstrained upper bound *loc_upper*. This artificial upper bound is the maximum room that is left to improve the current solution of $C$. Two flags characterize how overconstrained the resolution of the block is. *do_better* is

[6]Finding a feasible solution of a binate covering problem comes down to finding a true assignment of a conjunctive normal form, which is NP-complete.

[7]It was already used to produce the experimental results of [5].

[8]It can be infinite when it is not know whether the problem is feasible.

true iff we must improve the block's current solution so that $C$'s solution can be improved. *overcons* is true iff we forced a better upper bound. After having explored the (possibly smaller) search space of the block, we can decide whether the block has been solved, and whether we can abort the resolution of the partition.

## 5 Experimental Results

Caching all the subproblems generated during a branch-and-bound is definitely not effective, because the hit ratio is nearly zero. However caching the blocks generated by a partition is effective. Most of the time, only small blocks (say $10 \times 10$ covering matrices) are retrieved, but it happens that larger blocks are generated several times, reducing substantially the CPU time.

Table 1 shows the performances of our solver SCHERZO with and without the partition based pruning technique presented in Section 4. The upper (respectively lower) part of the table presents data for unate (respectively binate) covering problems. We have chosen examples that can be partitioned during the branch-and-bound resolution. Of course, this does not happen with all examples, but the sparser the covering matrix, the more likely partitions can be found. Table 1 shows that when the covering matrix can be partitioned into independent blocks during the branch-and-bound, caching the blocks and overconstraining the block resolutions can reduce the CPU time by a factor of up to 100.

Table 2 compares the performances of SCHERZO with an espresso-like [13] version of SCHERZO, i.e., without the improvements presented both in this paper and [5, 4]. The table clearly shows that the search space that must be explored to solve both unate and binate covering problems is dramatically reduced, and consequently the CPU time is cut by a factor of up to 1000.

## 6 Conclusion

This paper has presented some more insights in solving unate and binate covering problems. When solving covering problems with a branch-and-bound algorithm, quickly computing a good quality lower bound is a prime necessity. We investigated the tradeoff between the theoretical approximation ratio and the "practical" quality of two lower bound computation algorithms. We also presented a partition based pruning method that can substantially speed up the resolution. Experiences show that the results presented both in this paper and [5, 4] improve the state-of-the-art of unate and binate covering solver up to three order of magnitudes in speed.

| **Name** | **Without** | | **With** | | | |
|---|---|---|---|---|---|---|
| | **node** | **CPU** | **LU** | **XAT** | **node** | **CPU** |
| *mlp4L* | 5546 | 29.1 | 0 | 21 | 984 | 12.1 |
| *m4L* | 35386 | 62.0 | 4 | 151 | 3428 | 19.5 |
| *m2I* | 1301374 | 1076.2 | 501 | 2085 | 32692 | 149.6 |
| $max512L$ | 3939882 | 4924.4 | 21 | 326 | 6902 | 110.2 |
| *addm4L* | 1513341 | 4301.9 | 2 | 119 | 1114 | 21.7 |
| *expsI* | – | $> 2h$ | 412 | 4176 | 22877 | 414.6 |
| *count.b* | 43314 | 85.3 | 393 | 873 | 18280 | 64.7 |
| *des.a* | 144569 | 202.0 | 459 | 3743 | 26106 | 146.5 |
| $apex4.a$ | 1625529 | 937.3 | 6755 | 7298 | 33898 | 149.4 |
| $int13$ | 35971 | 189.3 | 65 | 196 | 16779 | 147.0 |
| $C880.b$ | 428711 | 1084.8 | 174 | 603 | 52187 | 234.8 |
| $int14$ | – | $> 2h$ | 2669 | 33869 | 475266 | 6584.3 |

The table gives the #hits on upper and lower bound (**LU**), the #hits on solved blocks (**XAT**), i.e., with status *SOLVED* and *UNSAT*, the number of **node**s in the search tree, and the **CPU** time in seconds on a 60 MHz SuperSparc (85.4 SpecInt).

Table 1. SCHERZO with and without the partition based pruning method.

## References


[1] R. Boppana, M. Halldórsson, "Approximating Maximum Independent Sets by Excluding Subgraphs", Bit **32**, pp. 180–196, 1992.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Acad. Pub., 1984.

[3] V. Chvátal, "A Greedy Heuristic for the Set-Covering Problem", *Math. Op. Res.*, **4**-3, pp. 233–235, Aug. 1979.

[4] O. Coudert, J.-C. Madre, "New Ideas for Solving Covering Problems", *32nd DAC*, pp. 641–646, June 1995.

[5] O. Coudert, "Two-Level Logic Minimization: An Overview", *Integration*, **17**-2, pp. 97–140, Oct. 1994.

[6] S. Devadas, A. Ghosh, K. Keutzer, *Logic Synthesis*, McGraw-Hill, 1994.

[7] J. F. Gimpel, "A Reduction Technique for Prime Implicant Tables", *IEEE Trans. Elec. Comp.*, **14**, pp. 535–541, June 1965.

[8] D. S. Johnson, "Approximation Algorithms for Combinatorial Problems", *J. Comp. Sys. Sci.*, **9**, 1974.

[9] E. L. Jr. McCluskey, "Minimization of Boolean Functions", *Bell Sys. Tech. Jour.*, **35**, pp. 1417–1444, April 1959.

[10] G. de Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[11] W. V. O. Quine, "On Cores and Prime Implicants of Truth Functions", *Am. Math. Monthly*, **66**, pp. 755–760, 1959.

[12] S. Robinson, R. House, "Gimpel's Reduction Technique Extended to the Covering Problem With Costs", in *IEEE Trans. Elec. Comp.*, **16**, pp. 509–514, Aug. 1967.

[13] R. L. Rudell, *Logic Synthesis for VLSI Design*, PhD thesis, UCB/ERL M89/49, 1989.

| Example Name | sol | Esp−like node | Esp−like CPU | Scherzo node | Scherzo CPU |
|---|---|---|---|---|---|
| *lin.rom.oc* | 113 | 7966 | 161.9 | 24 | 1.5 |
| *test*1 | 110 | 252837 | 1327.9 | 284 | 5.1 |
| *foutL* | 364 | 1273837 | 1665.6 | 13112 | 52.4 |
| *m*4*L* | 1335 | 545728 | 1854.6 | 3428 | 19.5 |
| *mlp*4*L* | 1316 | 4265632 | 24339.3 | 984 | 12.1 |
| *max*512*L* | 1087 | – | $> 4d$ | 6902 | 110.2 |
| *ocex*1 | 69 | – | $> 4d$ | 6555 | 192.0 |
| *max*1024 | 259 | – | $> 4d$ | 131968 | 10472.5 |
| *prom*2 | 287 | – | $> 4d$ | 27958 | 11947.0 |
| *ex*5 | 65 | – | $> 4d$ | 105916 | 17514.5 |
| *add*4 | 31 | 616545 | 899.3 | 622 | 7.1 |
| 5*xp*1.*b* | 12 | 27925 | 106.6 | 6500 | 9.2 |
| *count.b* | 24 | 1166613 | 6358.0 | 18280 | 64.7 |
| *e*64.*a* | 80 | – | $> 2h$ | 160 | 0.6 |
| *sao*2.*b* | 25 | – | $> 2h$ | 280 | 1.2 |
| *jac*3 | 15 | – | $> 4d$ | 290 | 6.9 |
| *des.a* | 942 | – | $> 4d$ | 26106 | 146.5 |
| *int*13 | 110 | – | $> 4d$ | 16779 | 147.0 |
| *apex*4.*a* | 776 | – | $> 2h$ | 33898 | 149.4 |
| *C*880.*b* | 63 | – | $> 4d$ | 52187 | 234.8 |
| *int*10 | 116 | – | $> 2h$ | 998 | 507.4 |
| *int*14 | 140 | – | $> 4d$ | 475266 | 6584.3 |

The table gives the minimum cost **sol**ution, the number of **node**s in the search tree, and the **CPU** time in seconds on a 60 MHz SuperSparc (85.4 SpecInt).

Table 2. Espresso-like vs. Scherzo.

```
function  SolvePartition(C, partition)
unsol ← ∅;
lower ← 0;
upper ← 0;
foreach  c ∈ partition {
    RetrieveBlock(c);
    /* c.status is UNSAT, SOLVED, LU, or NEW. */
    if  c.status = UNSAT return UNSAT;
    if  c.status = NEW then
        ComputeLowerAndUpperBound(c);
    /* c.status is now LU or SOLVED. */
    if  c.status = LU then
        unsol ← unsol ∪ {c};
    lower ← lower + c.lower;
    upper ← upper + c.upper;
    if  lower ≥ C.upper return PRUNED;
}
foreach  c ∈ unsol {
    unsol     ← unsol − {c};
    loc_upper ← min(upper, C.upper) − Σ_{c∈unsol} c.lower;
    if  c.lower ≥ loc_upper return PRUNED;
    do_better ← (loc_upper ≤ c.upper);
    overcons  ← (loc_upper < c.upper);
    c.upper   ← min(c.upper, loc_upper);
    upper     ← upper − c.upper;
    Solve(c);
    /* c.status is UNSAT, SOLVED, or PRUNED. */
    if  c.status = UNSAT return UNSAT;
    if  c.status = PRUNED {
        if  (overcons = TRUE)
            then  c.status ← LU;
            else  c.status ← SOLVED;
        if  do_better = TRUE return PRUNED;
    }
    /* Update the global upper bound. */
    upper     ← upper + c.upper;
}
Build C's minimum solution and return SOLVED;
```

Figure 6: Solving a partition.