Logic Minimization using Two-column Rectangle Replacement*

Søren Søe and Kevin Karplus

Board of Studies in Computer Engineering
University of California, Santa Cruz

ABSTRACT

This paper describes a technique for multi-level logic minimization on functions represented as if-then-else DAGs. We define the concept of Boolean matrices, and give formal definitions of blocks and rectangles and their meanings. We introduce a new heuristic *two-column rectangle replacement* for finding rectangle coverings of Boolean matrices. This heuristic is well suited for optimizing circuits for area, while controlling the delay. A slight variation of the heuristic optimizes with respect to delay. The results of using two-column rectangle replacement on if-then-else DAGs are reported for several benchmark examples.

1 Introduction

This paper is concerned with factoring and recognizing shared subexpressions in Boolean functions. Boolean functions are represented as Boolean matrices, and rectangles of these matrices represent either a factor of a function or a subexpression that can be shared among several functions. We use a technique we call *two-column rectangle replacement*. The rectangle replacement problem is a variant of Brayton's rectangle covering problem [1,2]. In both we find sets of rectangles that cover all the 1's of the Boolean matrix—rectangle replacement differs from rectangle covering in the way rectangles are replaced. In Brayton's rectangle covering, rectangles are replaced in parallel. In rectangle replacement and two-column rectangle replacement, we replace rectangles in sequence. Because the Boolean matrix changes after each replacement, the solution to the two problems may differ significantly.

Section 2 shows how to represent Boolean functions as if-then-else DAGs [3,4]. Two-column rectangle replacement does not rely on the if-then-else DAG representation—it could be applied any time that rectangle covering is useful.

In Section 3 the semantics of *blocks* and *rectangles* are given, and we show how to replace rectangles of a matrix with simpler rectangles. We finally define the rectangle replacement problem, which consist of finding and replacing rectangles in the right order. In Section 4, we use *two-column rectangle replacement* on a multi-level logic minimization system where functions are represented as if-then-else DAGs [3,4]. Two-column rectangle replacement consist of finding rectangles with exactly two columns in the matrix. These two columns have associated Boolean expressions, which will be combined into a new expression using an associative and commutative logic operator. We present two heuristic methods for selecting the order of replacement: one optimizes for area, and one optimizes for delay. Section 5 presents some results.

*This research supported by NSF grant MIP-8903555.

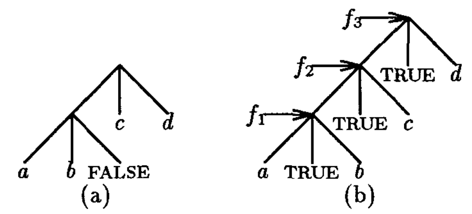

Figure 2.1: (a) If-then-else DAG for $abc + \neg ad + \neg bd$. The left branch from a node points to the **if**-part, the center to the **then**-part, and the right to the **else**-part. (b) Multiply-rooted DAG. The three expressions represented are $f_1 = a + b$, $f_2 = a + b + c$ and $f_3 = a + b + c + d$.

2 If-then-else DAGs

Representing Boolean functions with if-then-else DAGs is described in detail in earlier reports [3,4]. The if-then-else operator is the basic operator underlying if-then-else DAGs, with (**if** a **then** b **else** c) defined as $ab + \neg ac$ or, equivalently, $(a + c)(\neg a + b)$. We can define all Boolean functions of two variables using one if-then-else-operator and negation.

Definition 1: *An* if-then-else DAG *is a ternary directed acyclic graph in which each leaf is labeled with* TRUE, FALSE, *or a literal and each internal node has three out-edges pointing to the* **if**-, **then**-, *and* **else**-*parts. The meaning of a leaf node is the label on the node, and the meaning of an internal node is defined recursively as* (**if** meaning(**if**-part) **then** meaning(**then**-part) **else** meaning(**else**-part)).

If we eliminate trivial triples that have constants in the **if**-part or have identical **then**- and **else**-parts, we have four types of triples: AND-triples, where one of the **then**- and **else**-parts is FALSE, OR-triples, where one of the **then**- and **else**-parts is TRUE, XOR-triples, where the **then**-part is the negated of the **else**-part, and finally IF-triples.

In Figure 2.1(a) an if-then-else DAG for the expression $abc + \neg ad + \neg bd$ is shown. A network of functions is represented as a multiply-rooted DAG, where each function is associated with one root, Figure 2.1(b).

Local factoring methods can factor if-then-else DAGs [4], but the methods fail to give the global view needed to identify that two expressions can be identical even though they are represented differently. For instance, $a+b+c$ and $a+c$ might not be recognized as having a common subexpression, unless they happen to be transformed to a common form.

In this paper we explore rectangle replacement, a variant of rectangle covering [1,2], which has been useful for finding common subexpressions in sum-of-products minimizers, and see how it can be applied to if-then-else DAGs.

3 Boolean matrices, blocks and rectangles

A *Boolean matrix* is a two-dimensional matrix representing a logic expression or a set of logic expressions. Each row and each column in the matrix is associated with its own

OR-matrix	a	b	de	c(f+g)	cf	cg
$f_1 = a + b + de$	1	1	1	0	0	0
$f_2 = b + de + c(f+g)$	0	1	1	1	0	0
$f_3 = cf + cg$	0	0	0	0	1	1

Figure 3.1: Boolean OR-matrix for three functions f_1, f_2, and f_3. Note that the columns are not limited to simple AND-terms.

matrix type:	XOR	AND	OR
row from columns	$\oplus$	$\bigwedge$	$\bigvee$
block from rows	$\bigwedge$	$\bigvee$	$\bigwedge$

Table 3.1: Operators for determining the meanings of rows and blocks in matrices of different types.

expression. The row expressions are built out of some combination of the column expressions, using any associative, commutative operation to combine the column expressions into rows. An entry B_{rc} in the Boolean matrix takes the values 1, 0, and d depending on whether the expression for row r contains column c (1), doesn't contain it (0), or we don't care whether it contains it or not (d). Figure 3.1 shows a Boolean matrix for three functions—the matrix is referred to as an OR-matrix, since each row is the OR of the corresponding columns. We need to be able to talk about parts of the matrix as single entities, and so we define a block:

Definition 2: *A* block *of a Boolean matrix B is any subset R of rows and any subset C of columns in B.*

The meaning of a row or a block depends on the operation that relates the row and column expressions in the matrix.

Definition 3: *After setting each d independently to either 0 or 1, the* meaning of a row *in an OR-matrix is the expression obtained by or-ing together all the column expressions for columns that have a 1 in the row. That is,*

$$\text{meaning\_row}(r, C, B) = \bigvee_{c \in C, B_{rc}=1} c \ .$$

Definition 4: *The* meaning of a block *in an OR-matrix is the expression obtained by and-ing together the meanings of each row in the block. The operator should be a true Boolean operator, so that $abab = ab$. That is,*

$$\text{meaning\_block}((R, C), B) = \bigwedge_{r \in R} \text{meaning\_row}(r, C, B) \ .$$

The definitions for the meaning of rows and blocks in AND and XOR matrices are similar—we just change the operators according to Table 3.1. We require that both operators be commutative and associative.

In order to handle the don't-cares correctly when we define replacement of blocks, we define acceptable row expressions:

Definition 5: *Row r_2 is an* acceptable replacement *for row r_1 if for every setting s_2 of the d's in r_2 there exists a setting s_1 of the d's in r_1, such that the meaning of r_1 based on the setting s_1 is the same as the meaning of r_2 based on the setting s_2. Note that the rows do not have to use the same columns.*

In Figure 3.1 the first two rows of the matrix have two columns that have 1's in both rows. We call such combinations of rows and columns a *rectangle*:

OR-matrix	a	b	de	$c(f+g)$	cf	cg	$b+de$
$f_1 = a + b + de$	1	d	d	0	0	0	1
$f_2 = b + de + c(f+g)$	0	d	d	1	0	0	1
$f_3 = cf + cg$	0	0	0	0	1	1	0

Figure 3.2: Boolean matrix for the functions f_1, f_2, and f_3, after the new column $b + de$ has replaced the columns b and de.

Definition 6: *We say that a block (R, C) of a Boolean matrix B is a* rectangle *if for every $r \in R$ and $c \in C$, we have $B_{rc} \neq 0$.*

3.1 Replacing rectangles in a Boolean matrix

Replacing a rectangle means adding a new column to a matrix corresponding to some rectangle of the matrix, and replacing 1's in the original rectangle with the 1's in the new column, while preserving the meaning of important blocks of the Boolean matrix.

Lemma 1 (Replacement): *A rectangle of a matrix can be replaced by adding to the matrix a new column C_{new}, whose associated expression is the meaning of the rectangle. Putting a 1 in C_{new} in each row contained in the rectangle and changing the 1's in the rectangle to d's makes any row in the new matrix an acceptable replacement for the same row in the old matrix.*

The proof follows immediately from the definition of the meaning of a block and the definition of acceptable replacement, by setting d's in the rectangle before replacement equal to 1, and setting other d's to match the setting in the new block. Note that the lemma is essentially the correctness proof for MISII's cube extraction algorithm.

Replacing a rectangle reduces the number of 1's in the matrix by an amount we call the value of a rectangle:

Definition 7: *The* value *of a rectangle is the difference in the number of 1's in the matrix before and after replacement of the rectangle.*

As an example, consider the Boolean matrix shown in Figure 3.1. If we replace the rectangle consisting of the first and second row and the second and third column (value=2), we end up with the matrix shown in Figure 3.2. Rectangles that span more than one row and more than one column are particularly useful to replace, as the new column, in this case $b+de$, will then be explicitly used as a shared subexpression for the rows of the rectangle. Note that this technique unfortunately does not recognize the commonality of $c(f+g)$ and $cf + cg$. We are still looking for methods to find such common subexpressions at an acceptable cost.

3.2 The rectangle replacement problem

We define the *rectangle replacement problem* to be the problem of sequentially replacing rectangles of a Boolean matrix until all the rectangles have value zero or less. This termination condition guarantees that each row has at most one 1 in it, and rows that start out with 1's end up with exactly one 1. The order in which we replace rectangles affects the quality of the resulting representation for the functions. The rectangle replacement problem is to choose an ordering for the replacement of rectangles that minimizes the predicted area or delay for the circuit. Solving the rectangle replacement problem also solves the rectangle covering problem as we have covered all the 1's in the Boolean matrix.

After solving the replacement problem, we replace each row expression by the column expression for which the row has a 1. A column expression may contain a row expression as a subexpression, in which further substitution is done.

4 Two-column rectangle replacement

In this section we will show how we apply the rectangle replacement problem to minimization using if-then-else DAGs. The technique that will be described corresponds roughly to cube extraction in MISII.

Of the four different types of if-then-else triples three, the AND-, OR-, and XOR-triples, can be used for building Boolean matrices. Using De Morgan's laws we merge AND- and OR-expressions, and create two Boolean matrices: the OR-matrix for the combined AND- and OR-expressions, and the XOR-matrix for XOR-expressions.

The OR- and the XOR-matrix are built by traversing the if-then-else DAG from each root and looking for AND-, OR-, and XOR-triples. Whenever we find one, create a new row. If any of the children of the triple (its inputs) are triples of the same type, we recursively include them in the same row. The inputs to the expression become columns of the matrix.

4.1 Rectangle replacement algorithm

After creating a matrix we apply the rectangle replacement algorithm to it. The algorithm replaces rectangles of a Boolean matrix B sequentially by using two sub-procedures:

select_rectangle, which selects a rectangle based on a heuristic method described in Section 4.2,

replace_rectangle, which replaces a rectangle according to the replacement strategy presented in Section 3.1.

The algorithm itself is fairly simple:

```
replace_rectangles($B$) =
    while ($\exists$ rectangles with value>0 in $B$) do
    {
        rect = select_rectangle($B$)
        replace_rectangle(rect, $B$)
    }
```

When the algorithm terminates, each row contains exactly one 1. We create a new multiply-rooted if-then-else DAG by traversing the old one from the roots, replacing each sub-DAG that corresponds to a row with the column expression for which that row has a 1. We continue the traversal with the children of the column expression, so that all necessary replacements are done in one traversal.

4.2 Selecting rectangles

Because the if-then-else DAG representation forces n-ary associative operators to be represented as binary trees, we create new columns from exactly two existing columns. The two-column rectangle replacement heuristic is formulated as:

Two-column rectangle replacement: *As long as there are rectangles containing exactly 2 columns with value>0, replace the rectangle of greatest value. If two or more rectangles have the same value, choose the one in which the new column would have the earliest arrival time.*

By considering only two-column rectangles and only one rectangle at a time, we have reduced the problem of finding rectangles. In a matrix with n columns there are $\binom{n}{2}$ possible two-column rectangles and 2^n possible multi-column (prime [1,2]) rectangles. Hence, we can afford to enumerate all two-column rectangles and choose the best, whereas choosing a prime rectangle must resort to a limited enumeration.

As was noted in Section 3.1, rectangles spanning more than one row are particularly useful to replace. This is reflected in the two-column rectangle replacement method by choosing the rectangle with the highest value (the value of a two-column rectangle of all 1's equals the number of rows in the rectangle).

The primary goal of two-column rectangle replacement is to optimize with respect to area, but if two or more rectangles have the same value, we choose the one that results in the earliest arrival time of the new column expression. We have previously seen that the height of a DAG is a usable delay estimate for the final circuit [4], and so we use height as our arrival time estimator. Using the height of DAGs to break ties results in a primitive form of tree balancing.

Tree balancing can be carried a little further. By changing the two-column rectangle replacement method only slightly, we optimize for delay instead of area:

Two-column rectangle replacement, optimizing for delay: *As long as there are rectangles containing exactly 2 columns with value>0, replace the rectangle in which the new column would have the earliest arrival time. If two or more rectangles result in the same arrival time, choose the one with the greatest value.*

If we use the height of the DAGs as our delay estimate, this replacement method will balance the DAG. It is possible to use a weighted sum of height and value to sacrifice area for delay, or delay for area.

4.3 Expanding IF-expressions

In our main algorithm for minimizing a multiply-rooted if-then-else DAG, we first apply rectangle replacement to the XOR-matrix, then we create the OR-matrix and apply rectangle replacement to it. In creating the OR-matrix we have found that it exposes more sharing if we expand IF- and XOR-triples that are inputs to OR- or AND-expressions. An IF-expression is expanded to either $ab+\neg ac$ or $(a+c)(\neg a+b)$ depending on whether it is input to an OR- or AND-expression. However, if the expressions resulting from the expansion are used only for the original IF- or XOR-triple, we have lost the compact IF-expression without exposing sharing, and we should recover the original IF-expression [6].

5 Results

This section gives a summary of the benchmark results; a full tabulation has been printed in a technical report [6]. Forty benchmark circuits for the 1989 International Workshop on Logic Synthesis [5] were optimized using our two-column rectangle replacement heuristic. The results are reported after local factoring, two-column rectangle replacement, and technology mapping have been applied. Each input BLIF gate is factored, and each output DAG is factored after composing the gates.

Two-column rectangle replacement is done after all local factoring is finished. The resulting DAGs were first mapped with a crude mapper to AND, OR, XOR, and IF gates, using essentially the same traversal algorithm used to generate the rows of the Boolean matrices, but making sure that shared subexpressions were always represented by gate outputs in the output BLIF file. Then the resulting Boolean networks were mapped to the lib2 library of the benchmark set by MISII. The mapping was done with map; gp. Each network had a standard input drive and output load equivalent to the *inv2x* cell, as specified in the benchmark. The area and delay are as reported by MISII after mapping.

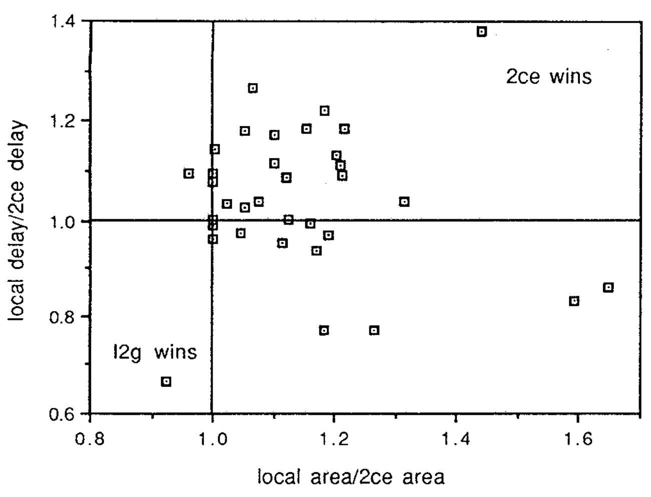

Figure 5.1: Comparison of the areas and delays of circuits minimized by local factoring (local) and by local factoring plus two-column rectangle replacement with expansion of IF-expressions (2ce).

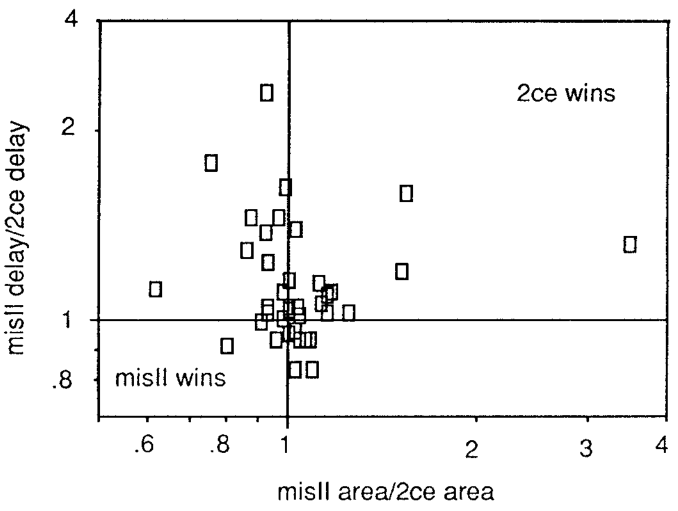

Figure 5.2: Comparison of the areas and delays of circuits minimized by local factoring plus two-column rectangle replacement with expansion of IF-expressions (2ce) and by MISII.

Figure 5.1 compares the mapped circuits produced by two-column rectangle replacement using expansion of IF-expressions with those produced by local factoring alone. We plot the ratios of the area on the x-axis, and the ratio of the delays on the y-axis. On the average we achieved a 10.6% reduction in area, and a 2.5% reduction in delay by using two-column rectangle replacement.

When optimizing for delay, the results show a 6.5% increase in area and a 6.3% reduction in delay when compared to the two-column rectangle replacement optimizing for area. When we optimize for delay, we change the order of rectangle replacement by choosing the two-column rectangle with the earliest arrival time. Unfortunately this may increase the total number of replacements needed in order to cover the Boolean matrix, which can increase the height of the DAG. Of the 40 benchmark examples we tried optimizing for delay, 9 resulted in an increase in delay and 3 even had an increase in height (up to 14.3% more).

Not expanding IF-expressions results in a 2.3% increase in area and a 0.5% reduction in delay as compared to the results achieved by expanding IF-expressions.

We have also implemented a prime rectangle replacement heuristic [6], which differs from Brayton's prime rectangle covering only in the way the rectangles are replaced—we do it sequentially whereas Brayton does it in parallel. The result shows that two-column replacement uses 1.2% less area, and produces circuits that are 0.8% slower than those achieved with prime rectangle replacement. In our implementation, the two-column replacement method is several hundred times as fast as prime rectangle replacement.

Finally Figure 5.2 compares the mapped circuits produced by two-column rectangle replacement using expansion of IF-expressions with those produced by running the standard MISII script with Release Version 2 of MISII. On the average we achieved a 3.5% reduction in area, and a 12.7% reduction in delay when compared to MISII.

6 Conclusions and future work

Our new heuristic for solving the rectangle replacement problem appears to be effective in finding common subexpressions. Two-column rectangle replacement, which was inspired by the structure of if-then-else DAGs, is particularly interesting as it combines finding common subexpressions with balancing operator trees.

We are still looking for better ways to select rectangles for replacement. When optimizing for area the highest value rectangle is not always the best one to replace next [6]. The optimal replacement when optimizing for area may be the one that covers all one's in the Boolean matrix with the fewest replacements. When optimizing for delay we have two problems: the height of the DAG is not a very good delay estimate, and the heuristic we are using does not always balance the DAG.

We hope to have a release of our system available for other researchers to experiment with by third quarter 1991.

References

[1] R. K. Brayton. Algorithms for multi-level logic synthesis and optimization. In G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits—Logic Synthesis and Silicon Compilation*, pages 197–247. Martinus Nijhoff Publishers, 1987.

[2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangle covering problem. In *ICCAD-87*, pages 66–69, Santa Clara, CA, November 1987.

[3] K. Karplus. Representing Boolean functions with If-Then-Else DAGs. Technical Report UCSC-CRL-88-28, Computer Engineering, Univ. of Calif., Santa Cruz, Santa Cruz, CA 95064, December 1988.

[4] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, March 1989.

[5] R. Lisanke. Logic synthesis and optimization benchmarks. Technical report, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709, 16 December 1988.

[6] S. Søe and K. Karplus. Logic minimization using block covering. Technical report, Computer Engineering, Univ. of Calif., Santa Cruz, Santa Cruz, CA 95064, 1991.