An Exact Gate Assignment Algorithm for Tree Circuits Under Rise and Fall Delays

Arlindo L. Oliveira
Cadence European Labs./IST-INESC
Lisboa, Portugal.
aml@inesc.pt

Rajeev Murgai
Fujitsu Laboratories of America, Inc.
Sunnyvale, CA, USA.
murgai@fla.fujitsu.com

Abstract

In most libraries, gate parameters such as the pin-to-pin intrinsic delays, load-dependent coefficients, and input pin capacitances have different values for rising and falling signals. The performance optimization algorithms, however, assume a single value for each parameter.

It is known that under the load-independent delay model, the gate assignment (or resizing) problem is solvable in time polynomial in the circuit size when a single value is assumed for each parameter [5]. In the presence of different rise and fall parameter values, this problem was recently shown to be NP-complete even for chain and tree topology circuits under the simple load-independent delay model [8]. In this paper, we propose a dynamic programming algorithm for solving this problem exactly in pseudo-polynomial time for tree circuits. More specifically, we show that the problem can be solved in time proportional to the size of the tree circuit, the number of choices available in the library for each gate, and the delay of the circuit. *To the best of our knowledge, this is the first pseudo-polynomial exact algorithm for the gate assignment problem for trees in the presence of different rise and fall delays.* We present a straightforward way of extending this algorithm to general directed acyclic graphs. We present experimental results on a set of benchmark problems using a standard commercial library and show that our algorithm generates provably optimum delays for 72 out of 76 circuits. We also compare our technique with two approaches traditionally used to solve this problem in the industry & academia and show that it is slightly better than these two. Interestingly, both traditional approaches also yield delays not far from the optimum.

1 Motivation

Most of the research in performance optimization, including almost the entire body of theoretical work, considers only a single value for each cell parameter such as pin-to-pin intrinsic delay, load coefficient, and input pin capacitance [13, 11, 5, 9, 6, 1, 14, 3, 2]. In reality, each cell in the library has different values for rising and falling transitions, sometimes very different. For instance, in one of our sub-micron technologies, the rise and fall intrinsic delays for a path in a simple gate differed by 45% and the load coefficients, by 100%! To bridge this gap between research and reality, most optimization tools approximate each cell parameter by taking either an average or the maximum of rise and fall values. Clearly, both these strategies for computing the circuit delay are approximations: the first is optimistic, and the second, pessimistic.

Recently, Murgai proved that certain problems in performance optimization that can be solved in polynomial time assuming a single value for each cell parameter become NP-complete with different rise and fall values [8]. These problems are:

1. the local fanout optimization problem with the net topology fixed, and
2. the gate assignment problem[1] for minimizing the circuit delay under the load-independent delay model. This problem remains NP-complete even for simple chain and tree circuit topologies.

However, these NP-completeness results used transformations from PARTITION, which is NP-complete but not in the strong sense [4], and can be solved exactly by a pseudo-polynomial time algorithm. So the possibility of solving these problems exactly under separate rise and fall delay values in pseudo-polynomial time was left open.

In this paper, we revisit the second problem, that of minimum-delay gate assignment under the load-independent delay model in the presence of separate rise and fall delay values, and propose an exact dynamic programming algorithm for solving it in pseudo-polynomial time for tree circuits.

The paper is organized as follows. In Section 2, we summarize the popular gate delay models in the context of rise and fall parameters. Section 3 revisits the gate assignment problem. Our dynamic programming algorithm is described in Section 4 along with the implementation details and data structures used. Section 5 presents the experimental results. Finally, we conclude with directions for future work in Section 6.

2 Gate Delay Models

The model used to calculate the delay through a gate (or cell) is of central importance in timing analysis and optimization.

Given a single-output gate (or cell) g, let $\delta(j,g)$ denote the delay from an input pin j of the gate g to the output of g. We will use g to denote the output of g as well. Two delay models are popular: load-independent and load-dependent. The **load** c_g refers to the cumulative capacitance seen at the output of g. It is the sum of the input pin capacitances $\gamma(p)$ of all the fanout pins p of g.

In the **load-independent delay model**, the delay from an input pin j of a gate g to the output of g, $\delta(j,g)$ is the intrinsic delay

$$\delta(j,g) = \alpha(j,g) \tag{1}$$

In the **load-dependent delay model**,

$$\delta(j,g) = \alpha(j,g) + \beta(j,g)c_g \tag{2}$$

Here,
$\alpha(j,g)$ = intrinsic delay from j to g,
$\beta(j,g)$ = drive capability or load coefficient of the path from j to g,
c_g = load capacitance at the output of the gate g.

The gate library specifies α and β parameters for all input-pin to output-pin paths within each gate and γ values for all the input pins. In general, $\alpha(j,g)$ and $\beta(j,g)$ are different for different input

[1]Murgai called it the gate resizing problem.

pins j. If g has a single input pin (e.g., buffers, inverters) or if α and β values are identical for all input pins, we will drop the argument j.

The above description assumes a single value for each parameter α, β, and γ. However, it is well-known that delays for the rising and the falling transitions can be quite different. In fact, every gate in the industrial gate libraries we have access to has different rise and fall delay parameter values. Quite often, these values are far off from each other. For instance, in one of our sub-micron technologies, the rise and fall α values for a path in a simple gate differed by 45% and the β values, by 100%! To handle this scenario, we use the subscripts r and f to denote rise and fall. For instance, $\alpha_r(j, g)$ denotes the intrinsic delay from input pin j to the output g when g switches from 0 to 1. Similarly, $\alpha_f(j, g)$ is the intrinsic delay when g makes a falling transition. We write these values as pairs: (α_r, α_f) and (β_r, β_f).

The gate delay parameters (α_r, α_f) and (β_r, β_f) are used to compute the **arrival times** at various gates and the delay through the circuit as follows. At each gate, both rise and fall arrival times are stored. The rise (fall) arrival time at a gate g denotes the maximum possible time it takes for a transition to travel from a primary input to the output of g and g makes a rising (falling) transition as a result. A topological traversal of the circuit from primary inputs to outputs is used to compute the rise and fall arrival times at each gate g using the rise and fall arrival times already computed at the fanin gates and the gate delays δ through g. An inversion through the gate should be considered appropriately while computing the times. For instance, since a falling transition at the input of an inverter generates a rising transition at its output, the fall arrival time at the inverter's input should be used to compute the rise arrival time at its output. The arrival time at a primary output is the maximum of the rise and fall arrival times at that output. The **delay of the circuit** is the maximum arrival time at primary outputs.

3 Gate Assignment Under Rise & Fall Delays

Gate assignment for minimizing the circuit delay is a fundamental problem in performance optimization of gate-level circuits. Ideally, each gate should be optimally sized during technology mapping. However, exact technology mapping is expensive in practice due to the large size of the technology library and due to the complex interaction between the gate being mapped and the unmapped portion of the logic. In addition, wire loads often cannot be estimated with sufficient accuracy during technology mapping to make the best choices for gate sizes. As a result, heuristics are used, which, among other things, may not select the best sizes for gates from a delay perspective [3]. This leaves scope for improving the circuit delay by resizing gates after technology mapping. Being an in-place optimization technique, gate assignment is also layout-friendly (i.e., it does not disturb the placement and routing of cells) and can be used during or after layout when more accurate wire load and wire delay information are available. Thus, gate assignment has become an important optimization problem in its own right.

The problem can be stated as follows. We are given a circuit composed of single-output cells from a cell-library. For each cell C_i, many different sizes $1, \ldots, k, \ldots$ are available in the library, each size having possibly different area, input pin capacitances γ, intrinsic delays α, and load coefficients β. Let C_i^k denote assigning size k to cell C_i. *The gate assignment problem is to select the size of each cell such that the circuit delay is minimized. We assume the load-independent pin-to-pin delay model*, in which the delay through a path within a cell is just the intrinsic delay α. Although simplistic, this delay model is gaining popularity with the advent of gain-based synthesis [12] in the presence of large, almost continuous-sized libraries.

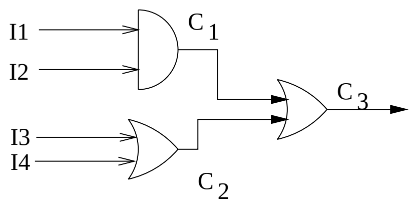

Figure 1: A circuit where both *same rise-fall* and *greedy* techniques yield sub-optimum solutions.

If only one value were to be used for each cell parameter (i.e., α), the problem can be solved optimally by a dynamic programming algorithm [5] as follows. Traverse the network gates in a topological order from primary inputs towards primary outputs. When a cell C_i is reached, the minimum possible arrival times at all its input pins are known. For each available size k of the cell C_i, compute the arrival time at the output of C_i^k using the arrival times at its input pins j and the pin-to-pin delays $\alpha(j, C_i^k)$ for the cell size k. Pick the size k^* that minimizes the arrival time at the output of C_i. Replace C_i with $C_i^{k^*}$. Continue the traversal and size selection until the primary outputs are reached.

If both α_r and α_f are specified, the circuit delay is given by max {circuit rise delay, circuit fall delay}, which is what we wish to minimize. In one strategy typically used in industry and academia for optimizing with different rise and fall delays, each pin-to-pin delay within a gate is approximated by a single delay value: $\alpha = \max\{\alpha_r, \alpha_f\}$. Under this approximation, the dynamic programming algorithm of [5], which was described in Section 3, is exact. Using these single pin-to-pin delay approximations, we apply the algorithm on the circuit to obtain the new gate sizes. Then, we do a delay trace on the modified circuit using actual rise and fall delays (α_r, α_f) to determine the true circuit delay. Let us call this strategy **same rise-fall**. In general, this strategy is sub-optimal. Consider the circuit shown in Figure 1. Let us assume rise and fall arrival times of 0 for all the primary inputs I_1 through I_4. Suppose there are the following two sizes for the AND gate:

1. with (α_r, α_f) of (7,3), and
2. with (α_r, α_f) of (4,6).

Similarly, suppose two sizes for the OR gate:

1. with (α_r, α_f) of (4,4), and
2. with (α_r, α_f) of (2,5).

In this example, for the sake of simplicity, we assume that for each gate size, both input pins have identical α_r values and identical α_f values, as shown above. In all, there are 8 possible ways to select gates for the circuit nodes shown. These 8 possibilities are shown in Table 1.

C_1		C_2		C_3		Delay
7	3	4	4	4	4	11
7	3	4	4	2	5	9
7	3	2	5	4	4	11
7	3	2	5	2	5	10
4	6	4	4	4	4	10
4	6	4	4	2	5	11
4	6	2	5	4	4	10
4	6	2	5	2	5	11

Table 1: Table of delays obtained for each possible gate choice.

With *same rise-fall*, the gate delays assigned to the two sizes of the *AND* gate are max{7,3} = 7 and 6 respectively, and to the two sizes of the *OR* gate, 4 and 5 respectively. Using these delay numbers, we can see that the size C_1^2 yields the minimum arrival time at the output of C_1 (the corresponding value = 6), C_2^1 at the output of C_2 (the value = 4), and C_3^1 at the output of the gate C_3 (the value = $\max\{6+4, 4+4\} = 10$). From Table 1, row 5, it can be checked that the circuit delay under this assignment is 10. However, from Table 1, it is easy to see that the optimum selection is obtained by choosing gates C_1^1, C_2^1 and C_3^2 (row 2 in the table), leading to the delay of 9.

Another natural strategy for using the dynamic programming paradigm is the following (we will call it **greedy**):

Maintain both rise and fall arrival times at each cell. For each cell, select the size that minimizes the maximum of rise and fall arrival times at that cell.

However, as shown in [8], *greedy* is also a sub-optimal strategy. Consider once again the circuit of Figure 1 and the gate sizes as shown above. By following this *greedy* strategy, one would select gate C_1^2 (with delays (4,6), since $\max\{4,6\} < \max\{7,3\}$), gate C_2^1 (with delays (4,4)), and gate C_3^1 (also with delays (4,4) – this can be seen from the fifth and sixth rows of Table 1). This leads to a circuit with the delay of 10, once again sub-optimal.

As this example shows, using this strategy we cannot decide locally at a gate the best size for it. We need to examine the fanouts as well. However, that may generate an exponential number of solutions by essentially enumerating all possible size selection choices in the circuit.

Murgai proved [8] that the problem of gate resizing with different rise and fall parameter values is NP-complete even under the load-independent delay model. The proof is based on transformation from PARTITION, a well known NP-complete problem [4]. PARTITION, stated as a decision problem, is as follows:
INSTANCE: A finite set A and a weight $w(a) \in Z^+$ for each $a \in A$.
QUESTION: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} w(a) = \sum_{a \in A - A'} w(a)? \tag{3}$$

However, interestingly PARTITION is not NP-complete in the strong sense. This distinction between strong and weak NP-complete problems is somewhat subtle, but important. Informally, a problem is NP-complete in the strong sense if its difficulty is not directly related to the values of the numbers used to describe an instance of the problem. Strong NP-complete problems are hard to solve even if the numbers that describe the instances of the problem are small. On the contrary, the complexity of weak NP-complete problems is directly related to the presence of large numbers in the instance description. NP-complete problems that are not strong can be solved in pseudo-polynomial time, i.e., in time polynomial in the largest number involved in the problem description.

One important example of a weak NP-complete problem is the PARTITION problem. PARTITION can be solved in polynomial time using a simple dynamic programming technique [4] that takes time polynomial in the sum of the numbers in A.

Since the gate assignment problem was proved NP-complete by transforming a weak NP-complete problem, there remains the possibility that the gate assignment problem is not, itself, NP-complete in the strong sense. In the next section, we partially settle this open problem by showing that a pseudo-polynomial time algorithm for solving the gate assignment problem exactly for tree circuits exists, thereby proving that the problem for tree circuits is not NP-complete in the strong sense. For the general directed acyclic circuits, the complexity of the problem still remains an open question.

4 Proposed Method

	1	2	3	4	5	6	7	8	9	10
1										
2										
3							1	1	1	1
4							1	1	1	1
5							1	1	1	1
6				2	2	2	1	1	1	1
7				2	2	2	1	1	1	1
8				2	2	2	1	1	1	1
9				2	2	2	1	1	1	1
10				2	2	2	1	1	1	1

Figure 2: Delay feasibility table T_1 for gate C_1. r is the column index and f the row index.

This section describes a dynamic programming approach that solves the gate assignment problem exactly in pseudo-polynomial time for a tree circuit. The algorithm is based on traversing the nodes of the tree circuit topologically from primary inputs to the output, computing the permissible fall and rise times for each node in the circuit, selecting that choice at the primary output which yields the minimum value of the circuit delay (which is the maximum of the rise and fall delays) from all permissible delays, and propagating this value backwards through the circuit and selecting the gate sizes. For simplicity, in the following exposition, we will consider the case where all the delays are integers, in some chosen unit.

Assume that, for each cell C_i in the circuit, several different sizes C_i^k are available in the library, each having different pin-to-pin delays. We assume the *load-independent pin-to-pin delay model*, in which the delay through a path within a cell is simply the intrinsic delay of the cell.

4.1 Delay Feasibility Table

The basic idea underlying the approach is to use a dynamic programming algorithm to compute, for each cell, the ranges of possible delays obtainable. For each cell C_i in the circuit, the permissible fall and rise times will be kept in a table T_i, the delay feasibility table. $T_i(r, f)$ will contain a value $k > 0$ if it is possible to achieve a rise arrival time of r and a fall arrival time of f by selecting the gate size k for that cell. If it is not possible to meet either r or f, then $T_i(r, f)$ will contain the value 0.

Upper limits on the required size of the table can be obtained from the circuit delay obtained with the greedy strategy described in Section 3. For the purposes of this analysis, we will assume the maximum circuit delay obtained with the greedy approach is $m = D$. This number represents an upper bound on the achievable circuit delay. For our example, as shown earlier, $m = 10$.

As an example, consider again the circuit in Figure 1. For gate C_1 in that figure, it is possible to achieve a rise time of 7 and a fall time of 3 by choosing gate C_1^1. It is also possible to achieve a rise time of 4 and a fall time of 6 by choosing gate C_1^2. According to the definition, $T_1(r, f)$ should take the value 1 for any pair (r, f) such that $(r \geq 7 \wedge f \geq 3)$. It should also take the value 2 for any pair (r, f) respecting $(r \geq 4 \wedge f \geq 6)$. Note that, for values of r and f satisfying both conditions (e.g., $r = 8, f = 8$), $T_i(r, f)$ can take either the value of 1 or 2, since both choices of gates achieve the desired arrival times.

For gate C_1, the delay feasibility table is shown in Figure 2. In this and the following figures, the value of r is used to index the column of the table and the value of f to index the row of the table. In this table, we chose arbitrarily gate C_1^1 for those delay values that can be obtained with either C_1^1 or C_1^2. The blank entries are assumed to contain the value 0 and denote delay infeasibility.

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										
4				1	1	1	1	1	1	1
5		2	2	1	1	1	1	1	1	1
6		2	2	1	1	1	1	1	1	1
7		2	2	1	1	1	1	1	1	1
8		2	2	1	1	1	1	1	1	1
9		2	2	1	1	1	1	1	1	1
10		2	2	1	1	1	1	1	1	1

Figure 3: Delay feasibility table T_2 for gate C_2.

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										
4							▨	▨	▨	
5									▨	
6									▨	
7									▨	
8									▨	
9									2	2
10								1	2	2

Figure 4: Delay feasibility table T_3 for gate C_3.

Similarly, for gate C_2, the table in Figure 3 is obtained.

For each cell C_i, the entries in table T_i are computed using a simple recursive relation. Let $S_i = \{C_j\}$ denote the set of gates that are the direct fanins of C_i. Let us also assume that the output of C_j is connected to the input pin j of C_i. We are interested in determining if a rise arrival time of r and a fall arrival time of f is possible at the output of C_i for some size assignment for all the gates in the sub-circuit rooted at C_i. We answer this by trying all gate sizes k at C_i one by one and then for each k checking the feasibility of appropriate rise and fall delay values at the fanin gates C_j, as shown in the following relation:

$$T_i(r,f) = k \text{ iff } \forall_{j\, s.t.\, C_j \in S_i}\ T_j(\widetilde{r_{j,k}}, \widetilde{f_{j,k}}) \neq 0 \quad (4)$$

where

- $\widetilde{r_{j,k}} = r - \alpha_r(j, C_i^k)$ and $\widetilde{f_{j,k}} = f - \alpha_f(j, C_i^k)$, if C_i is positive unate in the pin j, i.e., a rising (falling) transition at j causes a rising (falling) transition at C_i (e.g., $C_i = C_j + C_z$).
- $\widetilde{r_{j,k}} = f - \alpha_f(j, C_i^k)$ and $\widetilde{f_{j,k}} = r - \alpha_r(j, C_i^k)$, if C_i is negative unate in the pin j, i.e., a rising (falling) transition at j causes a falling (rising) transition at C_i (e.g., $C_i = C_j' + C_z$).
- $\widetilde{r_{j,k}} = \widetilde{f_{j,k}} = \min\{r - \alpha_r(j, C_i^k), f - \alpha_f(j, C_i^k)\}$, if C_i is binate in the pin j, i.e., a rising (falling) transition at j can cause both rising and falling (rising and falling) transitions at C_i (e.g., $C_i = C_jC_z + C_j'C_z'$).

Here, $\alpha_r(j, C_i^k)$ is the pin-to-pin delay from the input pin j to the output pin of C_i^k that corresponds to a rise transition on the output of the cell C_i^k caused by a transition on the input pin j, i.e., the input pin connected to the cell C_j. Similarly, $\alpha_f(j, C_i^k)$ is the delay that corresponds to a fall transition on the output of cell C_i^k caused by a transition on the pin j. Expression (4) states that there is a gate assignment for the sub-circuit rooted at the cell C_i, with the size k for the cell C_i, which can result in the rise and fall arrival times of r and f respectively at the output of C_i if and only if for each fanin gate C_j of C_i, there is a gate assignment for the sub-circuit rooted at C_j that can yield $\widetilde{r_{j,k}}$ and $\widetilde{f_{j,k}}$ as the rise and fall arrival times respectively at the output of C_j. The computation of $\widetilde{r_{j,k}}$ and $\widetilde{f_{j,k}}$ for the positive and negative unate cases is straightforward. For a binate pin j, consider a gate assignment rooted at C_j, which along with the size k at C_i results in a rise arrival time of r and a fall arrival time of f at C_i. Then, it must be the case that this assignment yields at the output of C_j, the rise arrival time of $r - \alpha_r(j, C_i^k)$ (considering the positive unate path) and also $f - \alpha_f(j, C_i^k)$ (considering the negative unate path). That is only possible if this assignment yields the rise arrival time at C_j of $\min\{r - \alpha_r(j, C_i^k), f - \alpha_f(j, C_i^k)\}$. Similarly, for the fall arrival time.

By sorting the cells in the circuit in a topological order from the inputs, the values of $T_i(r,f)$ can be computed by the following simple algorithm:

1. Select the next cell C_i in the circuit. Set $T_i(r,f)$ to 0, for all $r, f \leq m$.
2. For each possible choice C_i^k, do steps 3 and 4. If no more cells need processing, stop. Otherwise, go to 1.
3. For each cell C_j in the fanin of C_i, obtain $\alpha_r(j, C_i^k)$ and $\alpha_f(j, C_i^k)$ from the library data.
4. For all values of r in $\{1 \ldots m\}$ and f in $\{1 \ldots m\}$, do: if $\forall_j T_j(\widetilde{r_{j,k}}, \widetilde{f_{j,k}}) \neq 0$ then set $T_i(r,f)$ to k.

It is assumed that references to $T_i(r,f)$ with $r < 0$ or $f < 0$ will yield the value of 0.

To proceed with the example used above, we can now compute table T_3 for gate C_3 by applying the recursion in equation 4. T_3 is shown in Figure 4. For example, the element (9,9) in table T_3 is 2 because $T_1(9-2, 9-5)$ (i.e., $T_1(7,4)$) and $T_2(9-2, 9-5)$ (i.e., $T_2(7,4)$) – as shown by the dark squares – are both non-zero: see Figures 2 and 3. Recall that the size 2 for C_3 has a rise delay of 2 and a fall delay of 5 for both input pins, i.e., $\alpha_r(j, C_3^2) = 2$ and $\alpha_f(j, C_3^2) = 5$ for $j = 1, 2$. On the other hand, $T_3(8,9)$ is 0, because choosing either the size 1 or the size 2 for C_3 leads to infeasible delay assignments at the gate C_1. For instance, selecting C_3^2 implies that $T_1(8-2, 9-5)$ must be non-zero. But $T_1(6,4)$ is 0. Similarly, selecting C_3^1 implies that $T_1(8-4, 9-4)$ must be non-zero, which is not the case. Note that in this example, for simplicity, we used only positive unate gates.

Since C_3 is the circuit output, we stop the forward traversal of the circuit. Next, we select that entry (r,f) in T_3 which minimizes the maximum of $\{r_i, f_i\}$ over all non-zero entries $T_3(r_i, f_i)$. In our example, $T_3(9,9) = 2$ is the best entry, corresponding to the best achievable delay time of $\max\{9,9\} = 9$, obtainable by selecting the gate C_3^2 as the implementation for the cell C_3. The best gate assignment solution for the rest of the circuit can now be obtained by traversing the tables backwards. One finds that for gate C_3 we should select option C_3^2 (since $T_3(9,9)$ is 2), for gate C_2 one should select gate C_2^1 (since $T_2(9-2, 9-5) = T_2(7,4)$ is 1) and for gate C_1 one should select gate C_1^1 (since $T_1(7,4)$ is also 1).

Note that in this example we used equal rise delays and equal fall delays for all input pins to the gate output only for the sake of clarity. In general, different pins will have different delays, and that is taken into account in expression (4).

4.2 Complexity Analysis and Optimizations

From (4) and the algorithm proposed, a first analysis shows that the creation of table T_i requires a time proportional to $n_i p_i m^2$, where m is, as before, the dimension of the table, n_i is the number of possible choices for gate C_i and p_i the number of input pins for gate C_i.

It turns out, however, that computing the entire table is not necessary. In fact, for the purposes of delay optimization, only the boundary of the filled area[2] in the table needs to be computed, since no optimal solution will ever use cells properly inside the filled area. In Figure 3, the boundary is shown in bold. The computation of the cells in the boundary of the gate C_i can be performed by following the boundary of each gate in its direct fanin and filling in the corresponding entries in T_i. This procedure is illustrated in Figure 5. Since the maximum length of each boundary is no larger than $2m$, the computation of the boundary requires a time proportional to

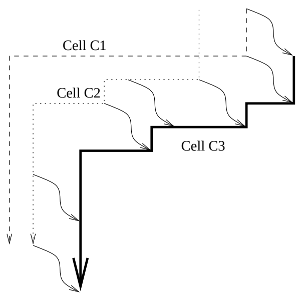

Figure 5: Computation of the boundary of the gate C_3 from the boundaries of its fanin gates C_1 and C_2.

$$n_i p_i m \tag{5}$$

With appropriate date structures, the memory requirements will also be proportional to this expression.

Consider now a library where the gate delays are not integers, but are given as floating point numbers. Assume that the precision of the gate delays is g (the granularity). For example, if the delays are given with a precision down to the hundredth of a nano-second, then $g = 0.01$. Additionally, assume that the value obtained for the circuit delay using the greedy approach described in Section 3 is D. Then, a table of size of $m = D/g$ will be required.

Summing expression 5 for all gates in the circuit, we obtain that the complexity of the algorithm is

$$O(\frac{RPGD}{g}) \tag{6}$$

where R is the maximum number of choices available for any gate in the library, P the maximum number of input pins in the gates, G is the total number of gates in the circuit, D is the circuit delay and g is the granularity.

Since a linear dependence in R, P and G is unavoidable for any gate sizing algorithm, the extra complexity is paid by the term $\frac{D}{g}$. This result was to be expected since in pseudo-polynomial time algorithms, the complexity is necessarily dependent on the size of the numbers involved, or, equivalently, on their precision.

4.3 Data Structures

Given the description of the algorithm and the note made in the previous section that only the boundaries of the filled part of the tables needs to be kept, there are several possibilities for the maintenance of the data stored in the tables T_i.

One possibility is simply to use a matrix for each table T_i. This matrix should be initialized to 0, and then only the boundaries need to be filled in. This solution is simple, but has the significant disadvantage that the memory requirements become proportional to $\frac{GD}{g^2}$ and that the initialization step may actually become dominant for small values of g, leading to a significant inefficiency.

[2]By filled area we mean the area of the tables that is filled with values different from 0.

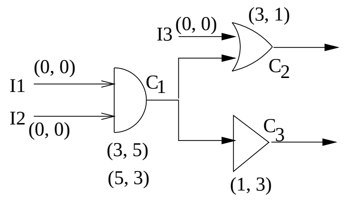

Figure 6: Sub-optimality for general DAGs

An interesting alternative is to use a sparse matrix data structure. If the accesses to the matrix elements are based on a hash table method, the memory requirements are now only proportional to $\frac{GD}{g}$, thus giving a much better asymptotic behavior. Regrettably, our experiences using a hash table based sparse matrix manipulation did indeed save memory, but at the expense of a substantial increase in the CPU time caused by the overhead in accessing the tables. This significant slowdown made the approach uninteresting, although it allows larger circuits to complete if given enough time.

We are currently addressing the problem of choosing an appropriate data structure that will yield the desired $O(\frac{GD}{g})$ memory usage without having such a significant impact on the CPU time. Conceptually, a solution that is efficient in both memory and CPU should be attainable, given the weak requirements imposed by the access patterns.

A more radical approach can be taken, reducing even further the memory requirements. Note that the boundary of each table T_i is totally defined by the exterior corners of the boundary of the filled area. For instance, the contents of the table shown in Figure 2 are totally defined by the values of the two corner cells $T_1(7,3) = 1$ and $T_1(4,6) = 2$. Using appropriate data structures, it may be possible to avoid the need to store the entire boundary. Note that, in the worst case, there may exist $O(m)$ external corners of the boundary, but, on average, the number of corners will be much smaller than m, leading to very significant savings in memory usage. However, whether it is possible to explore this property by using appropriate data structures remains an open question.

There are other significant details which have not been implemented in the current version of the algorithm and will speed up the algorithm and decrease the memory usage. One such optimization is based on the fact that the table size does not need to be the same for each node in the circuit, since the interesting part of the table T_i does not cover the whole range of indices from 1 to m. In fact, positions in the table indexed by coordinates smaller than the smallest possible delay in the direct fanins of C_i are useless, since the table contains only zeros in that region. On the other hand, positions in the table indexed by coordinates larger than the largest possible delay value for gate C_i are also useless, since they will never be used in the optimum solution. We believe these and related optimizations, when implemented, will reduce memory and CPU usage by at least on order of magnitude, making the approach very competitive with the simple greedy strategy described in Section 3.

4.4 Extension to Directed Acyclic Graphs

The algorithm proposed above is provably optimum only for tree circuits. For general combinational circuits, i.e., directed acyclic graphs (DAGs), it may not even yield a feasible solution. The reason is as follows. Consider the circuit of Figure 6 with the gate C_1 fanning out to two gates C_2 and C_3. Let C_2 and C_3 be the circuit outputs. Let C_1 have two sizes:

1. with (α_r, α_f) of $(3,5)$, and
2. with (α_r, α_f) of $(5,3)$.

C_2 has one size with (α_r, α_f) of (3,1) and C_3 has one size with (α_r, α_f) of (1,3). All inputs arrive at time (0,0). Our goal is to minimize the circuit delay. The delay feasible regions at the cell outputs are:

- C_1: $(r \geq 3 \wedge f \geq 5) \vee (r \geq 5 \wedge f \geq 3)$.
- C_2: $(r \geq 6 \wedge f \geq 6) \vee (r \geq 8 \wedge f \geq 4)$.
- C_3: $(r \geq 4 \wedge f \geq 8) \vee (r \geq 6 \wedge f \geq 6)$.

The best delay at the circuit outputs C_2 and C_3 is 6, corresponding to $(r, f) = (6,6)$. Since there is a single size for C_2 and C_3, to obtain these values, we need (3, 5) at the input of C_2 and (5, 3) at the input of C_3. Both these inputs are connected to the output of C_1. The constraint from C_2 requires that we select size 1 for C_1 whereas the constraint from C_3 mandates that we select size 2. Thus, (6, 6) is unrealizable! The problem is that C_1 is a multiple-fanout point, and the two fanout gates require selection of different sizes at C_1. Both sizes, although possible, *cannot be selected at the same time.*

There are several ways we can modify our algorithm for general DAGs. However, none of them is provably exact.

1. For DAGs, the best delay value ℓ computed at the circuit outputs from the delay feasibility tables is a lower bound on the true minimum delay possible by gate assignment. In this method, we compute the delay feasibility tables for each gate as before. We select the best delay values at the outputs. Then, we traverse the gates backwards (from primary outputs to primary inputs), selecting the size for each gate as dictated by the rise and fall delays propagated from the outputs, and propagating the (r, f) constraints to the fanins. When we hit a multi-fanout gate C, each fanout propagates different (r, f) constraints to C, requiring possibly different sizes for C. We pick the size that corresponds to the minimum value of $\max\{r, f\}$. After all the gates have thus been assigned sizes, we perform a delay trace on the modified circuit to obtain the true delay D of the resized circuit. If $D = \ell$, we know this algorithm has yielded the best possible delay.

2. Partition the DAG into trees, by cutting off at multiple fanout points (for instance). Order the trees topologically from inputs to outputs. For each tree in the order, apply the dynamic algorithm and select the gate sizes to minimize the delay at the output of the tree. Use these delay values while selecting the sizes for the gates in the trees later in the order. This is the heuristic widely used also in technology mapping.

Currently, we have implemented the first heuristic.

Note: If arbitrary gate replication is allowed in addition to gate assignment, the dynamic programming algorithm (resulting in the delay value ℓ) is provably optimum for general DAGs. To achieve this delay ℓ, when traversing the circuit backwards, for each multiple-fanout gate, create as many copies as the number of different sizes required by the fanouts.

5 Experimental Results

To evaluate the applicability of the method and the practical impact of our dynamic programming-based solution of the gate assignment problem, a preliminary implementation of the algorithm was developed and integrated with the SIS system [10]. In this section, we present the results obtained, both in terms of the final delay obtained for the circuit and the CPU times required to compute the solutions. We tested the algorithm on the set of circuits in the MCNC 91 benchmark, using Fujitsu's 0.5-μ full-strength technology library, with both unate and binate gates. For these experiments, we set the granularity g to 0.01 nanosecond.

We implemented three gate assignment algorithms:

- *same rise-fall* method, in which the pin-to-pin gate delay is approximated by a single delay value, and then the dynamic programming algorithm of [5] is applied. Finally, a delay trace on the modified circuit using actual rise and fall delays (α_r, α_f) yields the true circuit delay.
- the *greedy* method of Section 3: Under a topological traversal of the circuit, it selects for each gate the size that minimizes the maximum of rise and fall arrival times at that gate.
- the dynamic programming (DP) method of Section 4, as extended to general DAGs (see method 1 in Section 4.4).

The experiment was run on an Ultrasparc-60 that had 700MB of memory. Out of the 77 circuits present in this benchmark set, the DP algorithm was not able to complete on 1 example (C6288) due to memory limitations.

Out of the remaining 76 circuits, the DP method provably achieved the minimum delay on 72 of these circuits, and *same-rise-fall* & *greedy* did so on 55 and 63 circuits respectively. We know that a method generated the optimum delay value on a benchmark if it yielded a delay value identical to the corresponding lower bound ℓ generated by the forward pass of the DP method. To the best of our knowledge, this is the first time anyone has shown that the delays yielded by popularly used *same-rise-fall* and *greedy* algorithms, as well as by the DP method are exact for most benchmarks. This empirical evidence of the optimality of the DP method on almost all the circuits is very encouraging.

In 57 of the 76 circuits, the *same rise-fall*, *greedy*, and DP methods all obtained identical delays, 53 of which were provably optimum.

Statistics for the remaining 19 circuits are shown in Table 2. The numbers of circuit inputs, outputs and literals are shown in columns 2 through 4. The table also describes the results obtained on these circuits. The column 5 lists the lower bound ℓ on the optimum delay computed by the forward pass of the DP algorithm. This is the minimum delay any gate assignment algorithm can hope to achieve. Columns 6, 7, and 8 list the final delay values for the circuit where gate assignment was performed by *same rise-fall* (S), *greedy* (G), and the DP algorithm respectively. Columns 9 and 10 list the percentage improvements in delay by DP over S and G respectively. The last column shows the combined CPU times taken to solve the gate assignment problem with all the three methods.

The DP method resulted in provably minimum delay on all the 19 circuits, whereas *same-rise-fall* did so only on 2 and *greedy* on 10. So DP was better than *same-rise-fall* on 17 circuits and better than *greedy* on 9. The maximum delay improvement of DP over *same rise-fall* is 1.90% and that over *greedy* is 1.05%.

The performance of *greedy* was slightly better than *same-rise-fall*: on 11 circuits, *greedy* was better than *same rise-fall* and on 3 it was worse.

Finally, we note that all the algorithms are quite fast.

6 Conclusions and Future Work

In this work, we presented a pseudo-polynomial time algorithm for a problem that is known to be NP-complete, the gate assignment problem. *To the best of our knowledge, this is the first pseudo-polynomial exact algorithm for the gate assignment problem for tree circuits in the presence of different rise and fall delays.* Theoretically, it is an important result. We also presented a simple extension of the algorithm to general circuits.

A preliminary implementation of the algorithm was used to evaluate the performance of the commonly-used *same rise-fall* and *greedy* algorithms against our proposed dynamic programming solution. Although this implementation still uses simple data structures and suffers from inefficiencies in the memory usage, we were able to solve the gate assignment problem exactly for almost all circuits (72 out of 76) in a well-known benchmark set using the proposed dynamic programming technique, thus proving that this problem can be solved exactly for circuits of significant size.

These experiments have also shown that, in general, both *same rise-fall* and *greedy* approaches obtain results very close to the op-

					delay			% imp		
circuit	PI	PO	lits	ℓ	S	G	DP	DP/S	DP/G	CPU
9symml	9	1	277	1.83	1.84	1.84	1.83	0.54	0.54	1.3
C2670	233	140	2043	4.09	4.12	4.12	4.09	0.73	0.73	15.6
C3540	50	22	2934	5.22	5.23	5.23	5.22	0.19	0.19	18.9
C5315	178	123	4369	4.43	4.45	4.44	4.43	0.45	0.23	52.2
C7552	207	108	6098	3.56	3.57	3.57	3.56	0.28	0.28	52.0
alu2	10	6	453	4.38	4.42	4.38	4.38	0.90	0.00	3.9
cht	47	36	236	1.03	1.05	1.03	1.03	1.90	0.00	0.3
cm150	21	1	77	2.00	2.02	2.00	2.00	0.99	0.00	0.2
dalu	75	16	3067	4.25	4.27	4.25	4.25	0.47	0.00	40.0
lal	26	19	223	1.09	1.10	1.09	1.09	0.91	0.00	0.3
my_adder	33	17	257	6.26	6.27	6.26	6.26	0.16	0.00	10.1
pair	173	137	2420	3.19	3.20	3.19	3.19	0.31	0.00	10.9
rot	135	107	764	3.25	3.26	3.26	3.25	0.31	0.31	6.8
sct	19	15	164	1.41	1.42	1.41	1.41	0.70	0.00	0.3
t481	16	1	6823	4.14	4.14	4.15	4.14	0.00	0.24	32.9
too_large	38	3	1052	3.78	3.80	3.82	3.78	0.53	1.05	6.9
ttt2	24	21	341	1.23	1.24	1.23	1.23	0.81	0.00	0.5
vda	17	39	1423	1.94	1.94	1.95	1.94	0.00	0.51	3.2
x2	10	7	71	0.91	0.92	0.91	0.91	1.09	0.00	0.1

Table 2: Results for circuits where differences in delays were observed

PI (PO)	number of circuit inputs (outputs)
lits	number of literals in factored form
ℓ	lower bound on the minimum delay
S	*same rise-fall* method
G	*greedy* method
DP	the dynamic programming algorithm

timum, at least for the library used. This is despite the fact that the library contained gates with significantly different rise and fall delays. We believe this is due to the fact that even for circuits with a moderate number of levels (say 5), the imbalances in the individual rise and fall gate delays cancel out by the time the primary outputs are reached. Nevertheless, we were able to show for the first time that for the load-independent delay model in the presence of rise and fall delays, *same rise-fall* and *greedy* approaches work quite well, achieving exact results on 55 and 63 circuits respectively out of 76.

There are several interesting directions for future work. One obvious extension is to search for an exact pseudo-polynomial algorithm for general combinational circuits, which have gates with multiple fanouts. The other extension is to apply this method to the technology mapping problem. This should represent a relatively simple extension, as long as the load-independent delay model is assumed. Note that a generalization of the algorithm to the load-dependent delay model is not likely, since Murgai has recently proved that the gate assignment problem under the load-dependent delay model is NP-complete in the strong sense [7].

The memory usage currently represents the most significant bottleneck faced by the algorithm. It limits the applicability of the implementation to examples with tens of thousands of gates and libraries with very fine delay granularities. Additional work is needed on the data structures used in the manipulation of the tables.

References

[1] C. L. Berman, J. L. Carter, and K. F. Day. The Fanout Problem: From Theory to Practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, March 1989.

[2] P. Chan. Algorithms for Library-specific Sizing of Combinational Logic. In *DAC*, pages 353–356, 1990.

[3] O. Coudert, R. Haddad, and S. Manne. New Algorithms for Gate Sizing: A Comparative Study. In *DAC*, pages 734–739, 1996.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Mathematical Sciences Series. Freeman, 1979.

[5] Y. Kukimoto, R. K. Brayton, and P. Sawkar. Delay-Optimal Technology Mapping by DAG Covering. In *DAC*, pages 348–351, 1998.

[6] J. Lillis, C. K. Cheng, and T. T. Y. Lin. Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model. In *ICCAD*, pages 138–143, 1995.

[7] R. Murgai. On The Complexity of Minimum-delay Gate Resizing/Technology Mapping Under Load-Dependent Delay Model. In *IWLS*, pages 209–211, 1999.

[8] R. Murgai. Performance Optimization Under Rise and Fall Parameters. In *ICCAD*, pages 185–190, 1999.

[9] Lukas P. P. P. van Ginneken. Buffer Placement in Distributed RC-tree Networks for Minimum Elmore Delay. In *ISCAS*, pages 865–868, 1990.

[10] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[11] K. J. Singh. *Performance Optimization of Digital Circuits.* PhD thesis, UC Berkeley, December 1992.

[12] I. Sutherland and R. Sproul. The Theory of Logical Effort: Designing for Speed on the Back of an Envelope. In *Advanced Research in VLSI, University of California, Santa Cruz*, 1991.

[13] H. Touati. *Performance-oriented Technology Mapping.* PhD thesis, UC Berkeley, November 1990. UCB/ERL M90/109.

[14] H. Vaishnav and M. Pedram. Routability-Driven Fanout Optimization. In *DAC*, pages 230–235, 1993.