Chapter 6

Timing Driven Placement

6.1 Introduction

The purpose of an electronic circuit is to perform a useful function for a set of signals. An electronic circuit can be viewed as a *black box* which dynamically transforms a set of inputs into a set of outputs. The transformation between an input and an output is known as a transfer function *h* and is defined as

$$h(t) = \frac{output(t)}{input(t)} \tag{6.1}$$

Normally, the signals are time varying voltages, although sometimes they may be time varying currents.

An electronic circuit consists of components and wires. In the design process, wires are considered perfect conductors, and all the connection points of the components connected to a single wire are said to be at the same voltage. The time for a signal to propagate through a perfect conductor is limited only by the speed of light[1]. However, in the real world, conductors have finite conductance, capacitance, and inductance. In addition, capacitive and inductive coupling exists between different conductors. These *parasitic* devices limit the performance of electronic circuits and have serious ramifications for placement and routing algorithms.

Since most integrated circuits process time varying voltage signals rather than current signals, inductance parasitics may largely be ignored in the signal path. MOS (Metal

1. The speed of light in the given medium. The speed of light traveling on a printed circuit board is roughly 1/3 the speed of light in free space.

Oxide Semiconductor) digital circuits fall into this category. However, inductance parasitics must be considered if the current is switched. For example, it is common for power and ground signals to receive special treatment during placement and routing [33]. In this chapter, we will focus on time varying voltage signals for MOS circuits.

The connection between two gates of a MOS digital circuit can be modeled using the circuit in Figure 6.1. The signal propagates from the *driver* (output of a gate*)* to the *load*

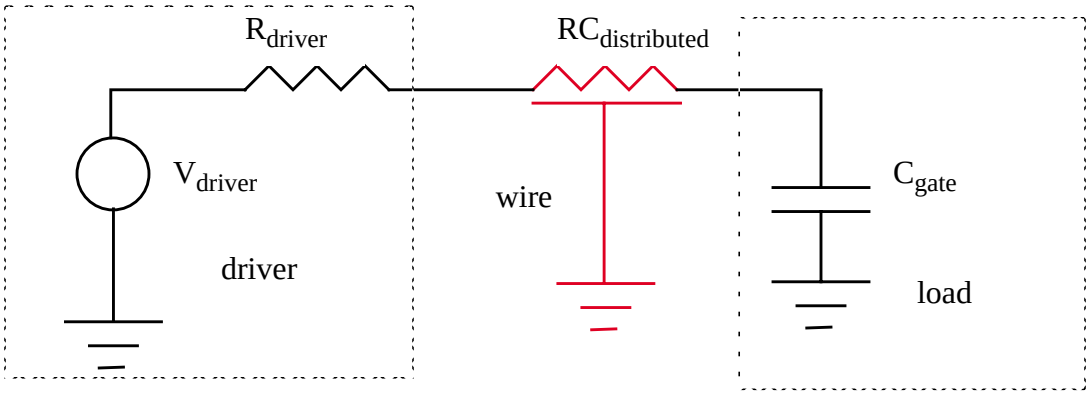

Figure 6.1 Model for interconnection of two MOS gates.

*(*input of a gate). In the simplified circuit, the driver is a voltage source with source resistance, and the load is a capacitor. The source resistance models the *drive strength* of the output transistor; the smaller the resistance, the stronger the transistor can drive the line. The load capacitor is equivalent to the gate capacitance of the MOS input transistors. The wire which connects them is a distributed RC (resistance capacitance) network. The driver and load are components; their values cannot be changed by the placement and routing algorithms. However, the wire's characteristics are determined by technology, and placement and routing. The parasitic capacitance of a wire segment is approximately that of a parallel plate capacitor. Its value is

$$C = \frac{\varepsilon A}{d} = \frac{\varepsilon l w}{t} \tag{6.2}$$

where ε is the permitivity of silicon dioxide, l is the length, w is width, and t is the thickness of the segment.

Since the width of capacitor is generally fixed to the minimum width design rule for the routing layer, and the thickness is set by the technology, the capacitor's value for a given routing layer is proportional to the length of the wire, or

$$C \propto l \, . \tag{6.3}$$

The resistance of a wire segment is given by

$$R = \frac{\rho l}{A} \tag{6.4}$$

where ρ is the resistivity of the material, and A is the cross sectional area. Sheet resistance is defined as

$$R_s = \frac{\rho}{t} \tag{6.5}$$

where t is the thickness of the wire segment. Substituting, we arrive at

$$R = R_s \cdot \frac{l}{w} = R_s \cdot \square \tag{6.6}$$

In both cases, the parasitic's value increases with the length of the wire. The goal of the placement and routing algorithm is to minimize the length of interconnections. For some materials, such as aluminum, the resistivity of the material is small enough to neglect the resistance of the wire for segment lengths encountered on the chip. For other materials, such as polysilicon, the resistivity is large ($R_s \approx 20 \ \Omega\text{square}^{-1}$) and cannot be neglected. In the first case, the distributed RC network simplifies into a single lumped capacitor. However, in the second case, the distributed RC network may not be simplified without losing accuracy. Since today's circuits are generally interconnected using only metal wire segments, we will concentrate on the simplified lumped capacitance model for

interconnect in this chapter[1].

Although zero length for all wires is an ultimate goal, it is impossible to achieve for any nontrivial circuit. Instead, the placement and routing algorithms need to minimize wire lengths on a set of *critical signal paths*. If the parasitic delays are large enough, the circuit may not function for these paths. Other signal paths in the circuit may not be affected by the addition of parasitic delays. In general, there is a lower bound T_{lb} and an upper bound T_{ub} on the time that a signal may propagate from the input to the output where the circuit still functions,

$$T_{lb} \le t_r \le T_{ub} \tag{6.7}$$

The delay from input to output can be decomposed into two pieces: component or gate delay and parasitic delay. If the gate delay is subtracted from the total path delay, the time bounds $\left(\hat{T}_{lb}, \hat{T}_{ub} \right)$ or *slack* for the wiring parasitics can be calculated.

$$\hat{T}_{lb} \le t_r \le \hat{T}_{ub} \tag{6.8}$$

Figure 6.2 shows an example of a digital logic circuit. There are three inputs (A, B, C) and two outputs (D, E). The fanout of a gate is the set of gates whose input is connected to

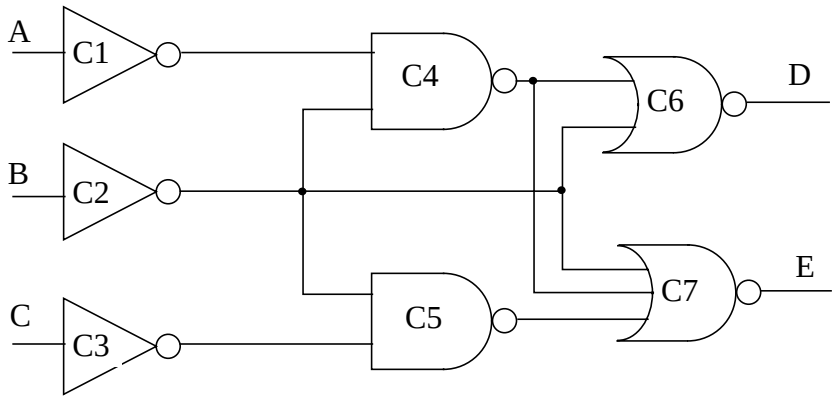

Figure 6.2 Example circuit.

1. Connections between layers known as vias also have resistance and capacitance. For most technologies, they may be neglected. An exception is anti-fuse FPGA technology.

the output of the gate. Similarly, the fanin of an input pin of a gate is the set of gates whose outputs are connected to that input pin. The transitive fanout of a gate is defined recursively by the repeated application of the fanout function. It is equivalent to the set of gates reachable from the output; that is, a path exists from the output to any gate in the set. For example, C4 is in the fanout of C1 whereas C4, C6, and C7 are in the transitive fanout of C1. The transitive fanin is defined analogously except that it defines the set of gates reachable from an input. In Figure 6.2, the transitive fanin of C7 pin 3 is C5, C2, and C3.

The gate and wiring delays for this circuit may be represented using a timing graph as shown in Figure 6.3. The timing graph is a weighted directed acyclic graph (*dag*). The nodes of the graph represent the signal pins of the circuit. The edges of the graph connect the pins. Each edge has a weight corresponding to the propagation delay. There are two types of edges: internal and external. The internal edges are signal paths which exist between pins of a gate. The weights of these edges are fixed by technology and device physics. The external edges denote paths created by the signal network. The weight of these edges depends on the length of the interconnect. The goal of the placement and routing algorithm is to satisfy the time bounds for all signal paths through the circuit.

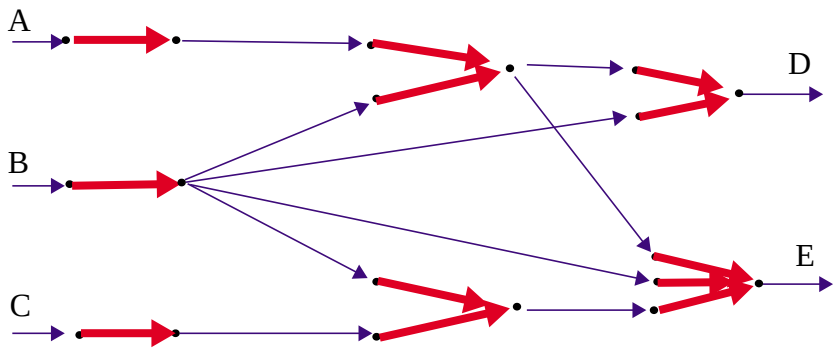

Figure 6.3 Timing graph for example circuit of Figure 6.2. The nodes of the graph are the signal pins. The edges of the graph connect the pins. There are two types of edges. The thick edges are signal paths through the gates whereas the thin lines denote the signal nets.

Often the designer knows the time constraints between a primary input pin and a primary output pin (such as B and E in Figure 6.3) but does not know the gates or nets of the paths between them. In Figure 6.3, there are three unique paths between pins B and E. The longest time among the three paths will set the upper bound on the time delay. The shortest time sets the lower bound. However, some of these paths may be logically or temporally incompatible and are therefore, *false paths*. A timing constraint for a false path is meaningless since the path can never be switched or *sensitized*. A false path unnecessarily constrains the placement problem.

For the example shown in Figure 6.4, three types of timing analysis are possible for the upper bound. The first analysis uses a modified form of the PERT[1] longest path algorithm [61]. It simply adds the longest delays to find the worst case path regardless of the logic state. In this case, the time delay is the sum of the two 20 ns delays. The second type of analysis, known as *static sensitization*, checks the logic state of each path to see if it is feasible. In the example, the inverter forces one mux to be in the select 0 state and one mux to be in the select 1 state. This sets the maximum delay along the data path to be 30 ns. The select path is now the longest path with a delay of 39 ns. The third analysis known as *dynamic sensitization* allows the logic states to change. It would discern the possibility

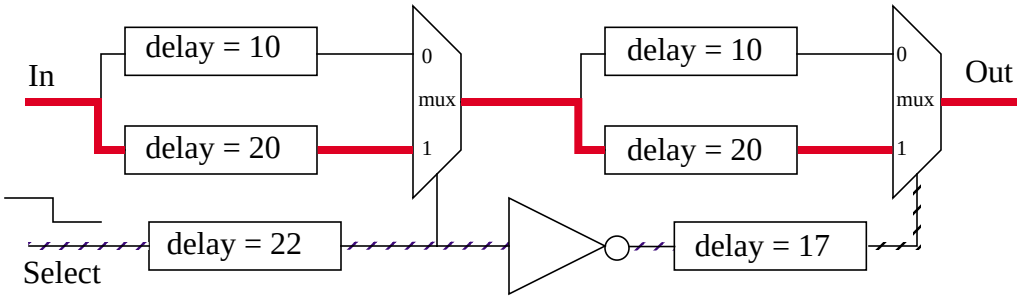

Figure 6.4 Typical example of a false path. PERT analysis yields solid path. Static sensitization finds the hatched path. Dynamic sensitization discovers that both paths may propagate together. From [7].

1. PERT is an acronym for “program evaluation and review technique.” This algorithm originated in operations research. It find the longest critical path in a job scheduling problem.

for the input signal to propagate through the two 20 ns delays if both the input and select line are switched at the same time. An algorithm has yet to be published which can guarantee to find the longest dynamically sensitized path. However, much progress has been made in synthesizing circuits which avoid these problems [110][188]. The tightest known upper bound on the true path delay is known as the *viable path* [147]. For most circuits, it coincides with the longest statically sensitized path.

In this work, a PERT-like algorithm is augmented to understand rising and falling transitions. The user may remove any false paths by enumerating them in a file. This is similar to the TA timing analyzer which allowed the user to add *delay modifiers* to indicate that a path was not possible [91]. The false paths may be obtained by using external timing analyzers.

6.2 Previous Work

Many timing driven placement schemes have been proposed [8]. The previous work can be divided into two categories: net-based [23][55][71][72][88][99][143][224][228] and path-based algorithms [53][63][85][98][145][210][211][218][220]. In a net-based algorithm, the timing delays are partitioned into constraints for the individual nets of the critical path. Early systems used a critical weight for each net which influences the behavior of the placement algorithm [23][55]. Dunlop et al. achieved acceptable designs by updating the net weights in an iterative loop consisting of a placement, routing, and timing analysis [55]. Burstein and Yousseff presented a hierarchical method which used timing analysis to update the net weights. Another approach uses timing analysis to determine precise bounds on the net delay and translates this information into constraints for the lengths of all nets [88][241]. If the constraints are met during placement, the timing specifications are guaranteed to be met. Recent systems use convex programming to translate the timing constraints into a set of upper bounds for net wire lengths [71][72]. Another method added net length constraints to the mincut placement technique [224].

The failure to consider the interactions between nets is a problem with net-based algorithms. Minimizing the delay in one net may create an excessive delay in another. It is the sum of the delay along a path which matters. Marek-Sadowska and Lin proposed the first path-based placement algorithm [145]. They used timing analysis to determine weights for the *rectilinear distance facility location* (RDFL) problem. These weights included a component which models the path length between cell instances.

In order to obtain accurate timing behavior and achieve better global solutions, path-based timing analysis must be performed *dynamically* during placement. Jackson and Kuh used linear programming (LP) and a path-based model which considers internal cell delays, interconnect, and pin capacitances to determine cell placement [98]. However, the complexity of the resulting linear program restricts its use to small problems. Partitioning must be performed to divide the placement problem into a manageable size for the LP solver. In addition, the I/O cells must be fixed to find a feasible placement. Srinivasan introduced a *reduced forest* of timing constraints to control the growth of timing equations in a nonlinear program [211]. While this expedited the execution time compared with the linear program and eliminated the need for an initial feasible placement, the cost function used the squared wirelength metric rather than the Manhattan metric. The squared wire length metric is known to be inferior [210]. Sutanthavibul and Shragowitz proposed a hierarchical constructive placement algorithm which analyzed the M longest and shortest paths to determine appropriate net weights [218]. This algorithm uses score functions which are nonadditive functions of their arguments. This case has shown that finding a set of M longest or shortest paths is NP-hard [243]. Sutanthavibul and Shragowitz resorted to an approximation algorithm based on Asano's algorithm which has a time complexity $O(m\log m + Mn)$ [5]. Hasegawa augmented force directed pairwise relaxation (FDPR) to handle path constraints [85]. In this case, the paths must be explicitly supplied by the user.

The use of timing driven placement with simulated annealing was first suggested for gate arrays by de Forcrand et al.[63] but the paper did not mention whether the time constraints were incorporated into the cost function. Donath et al. used simulated annealing in a hierarchical manner to place segments (the netlist is partitioned into segments) using complete path delays [53]. Timing analysis was used to produce parameterized delay equations for the paths. However, there was not an explicit limit on the number of delay equations that could be generated (in principle, an exponential number of paths can be generated), or a provision for handling false paths. The final solution quality depended on the quality of the partitioning step. Swartz and Sechen added timing critical paths to the simulated annealing cost function by summing the wire lengths of the nets in each critical path [220]. The timing driven placement algorithms presented in Section 6.3 extend this work.

6.3 Algorithm

We now present the algorithms which have been implemented in the TimberWolf place and route system. TimberWolf supports the following types of timing constraints:

- critical path using wire length constraints.
- matched critical path using wire length constraints.
- critical path using timing constraints.
- matched critical path using timing constraints.
- critical path analysis using pin pair constraints.
- matched critical path analysis using pin pair constraints.

6.3.1 Critical Path Using Wire Length Constraints

The simplest form of a timing constraint is one on the total wire length of a user specified critical path. For each critical timing path, the user supplies an upper and lower bound on the length of the path. The penalty assigned for a path p is the amount the length deviates from satisfying the bounds:

$$(p) = \begin{cases} length(p) - upperBound(p) & \text{if} \quad length(p) > upperBound(p) \\ lowerBound(p) - length(p) & \text{if} \quad length(p) < lowerBound(p) \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

where the length of a path p is the sum of the half-perimeter wire length of all the nets n in the path:

$$length(p) = \sum_{\forall n \in p} S_x(n) + S_y(n) \tag{6.10}$$

The total penalty is just the sum over all the specified critical paths:

$$P_T = \sum_{p=1}^{N_p} P(p) \tag{6.11}$$

Since the simulated annealing algorithm ensures that the sum of the lengths of the individual nets in the critical path meet the given bounds, there is no need for the user to partition the path length between the individual signals of the path. Previously reported systems have used net weights on individual nets in an attempt to achieve timing driven placement. However, this is the first cell placement algorithm which features critical-path driven placement. This method is superior to net weighting techniques because it overcomes the partition problem and reflects more accurately the true timing constraints to be satisfied.

6.3.2 Matched Critical Path Using Wire Length Constraints

On occasion, it is desirable to match the lengths of two paths. For example, the paths from the bond pad to the differential input pair of an operational amplifier should be matched to avoid offset errors. The user specifies a tolerance for the mismatch in path length. In this case, we assign the penalty for a pair of paths p to be:

$$(p) = \begin{cases} match(p) - tolerance(p) & \text{if} \quad match(p) > tolerance(p) \\ 0 & \text{otherwise} \end{cases} \tag{6.12}$$

where the match is defined as

$$match(p) = \left| length(p_1) - length(p_2) \right| \tag{6.13}$$

where p_1 and p_2 are the two paths comprising p.

6.3.3 Critical Path Using Timing Constraints

Although length constraints have been successfully applied to the timing design of several microprocessors, this method fails to account for differences in drive strength. The drive strength differences may be expressed using the driver source resistance. The propagation time for a signal path must be redefined as follows:

The arrival time T_a for a path *p* is the summation of all the net delays D_n for the path.

$$T_a(p) = \sum_{\forall n \in p} D_n \tag{6.14}$$

The delay for a single net *n* is the sum of the intrinsic gate delay T_n associated with the driver of the net *n*, and the product of the equivalent driver resistance R_n, and the total load capacitance seen by the driver C_n.

$$D_n = T_n + R_n C_n \tag{6.15}$$

The total capacitance for a net has two components: gate input capacitance and parasitic capacitance.

$$C_n = C_{G_n} + C_{p_n} \tag{6.16}$$

During placement, we can estimate the parasitic capacitance using the half-perimeter bounding-box metric:

$$C_{p_n} = C_{L_x} S_x(n) + C_{L_y} S_y(n) \tag{6.17}$$

Substituting Equation 6.16 and Equation 6.17 into Equation 6.15, we get:

$$D_n = T_n + R_n C_{G_n} + R_n [C_{L_x} S_x(n) + C_{L_y} S_y(n)] \quad (6.18)$$

We can precompute the terms in the summation which do not depend on wire length by defining:

$$K_p = \sum_{\forall n \in p} [T_n + R_n C_{G_n}] \quad (6.19)$$

This results in the following simplified equation for the arrival time:

$$T_a(p) = \sum_{\forall n \in p} R_n [C_{L_x} S_x(n) + C_{L_y} S_y(n)] + K_p \quad (6.20)$$

The penalty due to timing for a path p is given by,

$$(p) = \begin{cases} T_a(p) - T_{ru}(p) & \text{if} \quad T_a(p) > T_{ru}(p) \\ T_{rl}(p) - T_a(p) & \text{if} \quad T_a(p) < T_{rl}(p) \\ 0 & \text{otherwise} \end{cases} \quad (6.21)$$

where the user has specified an upper (T_{ru}) and lower bound (T_{rl}) on the required arrival times.

6.3.4 Matched Critical Path Using Timing Constraints

Matching critical paths when using the timing constraints is similar to the length constraint case. Again, the user specifies a tolerance for the mismatch in path delay. In this case, we assign the penalty for a pair of paths p to be:

$$(p) = \begin{cases} match(p) - tolerance(p) & \text{if} \quad match(p) > tolerance(p) \\ 0 & \text{otherwise} \end{cases} \quad (6.22)$$

where the match is now defined as the difference in arrival times,

$$match(p) = |T_a(p_1) - T_a(p_2)| \quad (6.23)$$

where paths p_1 and p_2 comprise the pair p.

6.3.5 Critical Path Analysis Using Pin Pair Constraints

Often the users do not know which paths are critical through the circuit. Instead, they are concerned with the timing delays between the primary inputs and primary outputs of the integrated circuit. Between a primary input pin and a primary output pin there may be many unique paths. The shortest non-false path will set a lower bound on the time delay whereas the longest non-false path will determine the upper bound. The timing delay has a parasitic element due to the wiring between components as shown in Equation 6.20. Different placements yield different wiring and thus timing delays. A particular path between two pins, for example, may be the longest path in one placement but the shortest path in another placement. Therefore, it is important to optimize all non-false paths between the two pins to ensure that timing specifications are met.

Algorithm Simulated-Annealing-With-Timing(M,K)

```
1   {FP} ← readFalsePaths()            /* read list of false paths from user */
2   buildTimingGraphs (P, G_n)
3   do
4       do
5           j = generate(i)
6           if accept(ΔC, T) then
7               i = j
8       until cost is in equilibrium
9       for each  = (s, t) ∈ P do      /* P is the set of all pin pairs */
10          if  _lb(s, t) > 0 then     /* a nonzero lower bound exists */
11              {T_a} ← findMShortestPaths (s, t, M)
12          {T_a} ← findMLongestPaths (s, t, M)
13      discard all but the K costliest paths
14      reduce(T)
15  until cost cannot be reduced any further
```

Figure 6.5 Modified simulated annealing algorithm with timing analysis. After the cost has reached equilibrium, a new set of timing constraints is generated. An upper bound of *M* such constraints are generated for each pin pair.

In order to achieve timing-driven placement with critical path analysis, we modify the simulated annealing algorithm as shown in Figure 6.5. The parameter *K* controls the number of paths to be monitored during a single execution of the outer loop. It allows the user to control the trade-off between accuracy and execution time. The first step of the algorithm reads false paths designated by the user. In the second step, a timing graph for each specified pin pair is constructed. After each iteration of the outer loop, a new set of paths to be monitored is created in lines 9 through 13. For each specified pin pair, the *M* shortest paths are found if a nonzero lower bound has been specified. In addition, the *M* longest paths are found by negating the edge weights of the timing graph and running the *M* shortest path algorithm. This is possible since the timing graph is a dag. These paths are now treated as time critical paths and Equation 6.21 is used to determine the cost. In line 13, all

```
Algorithm buildTimingGraphs (P, G_n)      /* P is the set of all pin pairs; G_n is the netlist graph */
 1   for each d = (i, j) ∈ P do           /* pin i is the input; pin j is the output of the pin pair d */
 2       FO_i ← transitiveFanout (i)      /* calculate the transitive fanout for input pin i */
 3       FI_j ← transitiveFanin (j)       /* calculate the transitive fanin for output pin j */
 4       N ← FO_i ∩ FI_j                  /* nodes which are in a path from i to j must be in intersection */
 5       for each vertex u ∈ V[G_n] − {s} do /* use bfs of netlist graph to build delay graph */
 6           visited[u] ← FALSE           /* initialization of the entire netlist */
 7       V[G_d] ← {i}                     /* initialize the set of nodes in the delay graph */
 8       E[G_d] ← { }                     /* initialize the set of edges in the delay graph to the empty set */
 9       Q ← {i}                          /* initial a queue with initial element i */
10       while Q ≠ { } do                 /* breadth-first search of netlist graph */
11           u ← head[Q]                  /* look at the element at the front of the queue */
12           for each v ∈ Adj[u]          /* must have a delay path from u to v */
13               if u ∈ N and u is an output and v ∈ N and (u, v) ∉ FP then
14                   V[G_d] ← V[G_d] ∪ {v}
15                   E[G_d] ← E[G_d] ∪ {(u, v)}
16                   if visited[v] = FALSE then
17                       visited[v] ← TRUE
18                       Enqueue(Q,v)     /* put v at the end of the queue */
19           Dequeue(Q,u)                 /* remove u from the front of the queue */
20       removeCycles (V[G_d], E[G_d])    /* remove any cycles so we have a DAG */
```

Figure 6.6 Algorithm which builds the delay graph between sets of input-output pin pairs.

but the costliest *K* time critical paths are discarded. Figure 6.6 describes the algorithm used to build the timing graphs for each of the specified pin pairs. The procedure is a modified breadth-first search algorithm (lines 5-19 inclusively). The algorithm prunes unnecessary nodes by admitting only nodes which are in the intersection of the transitive fanout of the source pin and transitive fanin of the target pin (lines 2-4 inclusively). Only these nodes can be reached from both source and target; hence, these nodes must exist on a path between the source and target. In line 13, specified false paths are eliminated from the graph. Cycles in the graph are detected and eliminated in line 20.

The transitive fanout algorithm is shown in Figure 6.7. It is simply a breadth first search modified to search input pins. The transitive fanin procedure proceeds in a similar manner from the input pin using the back edges of the graph.

Algorithm transitiveFanout(*s*)

```
1   for each vertex u ∈ V[G_n] − {s} do
2       visited[u] ← FALSE      /* initialization */
3   FO ← { }                    /* initialize fanout set to the empty set */
4   Q ← {s}                     /* initial a queue with source */
5   while Q ≠ { } do            /* breadth first search */
6       u ← head[Q]             /* look at the element at the front of the queue */
7       for each v ∈ Adj[u]     /* must have a delay path from u to v */
8           if visited[v] = FALSE and v is an input then
9               FO ← FO + {v}
10              visited[v] ← TRUE
11              Enqueue(Q,v)
12      Dequeue(Q,u)
```

Figure 6.7 Algorithm for calculating the transitive fanout of the input pin. The algorithm for calculating the transitive fanin of the output pin is similar except that the queue is initialized with the output pin in step 4 and in step 7 the back edges are traversed instead. In addition, in line 8 only the outputs are visited.

The timing graph construction routine creates a directed acyclic graph (dag). It is important that the graph be a dag; the dag's special properties are exploited in the *M* short-

est path algorithms. It allows the use of Dreyfus's method to compute the M shortest paths which has time complexity $O(Mn\log n)$ [54][128]. This is in contrast to Yen's method for general networks with time complexity $O(Mn^3)$ [242].

```
Algorithm removeCycles (V[G], E[G], s)
 1   {cycles} ← findCycles (V[G], E[G], s)
 2   for each cycle ∈ {cycles} do
 3       outputsOnly = TRUE
 4       for each node  n ∈ {cycle} do
 5           if n is not an output then outputsOnly = FALSE
 6       if outputsOnly then                  /* detected a wired OR connection */
 7           remove an edge between outputs that does not include source or target
 8       else                                 /* find the longest feedback connection */
 9           largestJump ← 0
10           for each edge (u, v) ∈ cycle do
11               if u is an output and v is an input then
12                   startDiff = startTime[u] – startTime[v]
13                   if |startDiff| > largestJump then
14                       largestJump = |startDiff|
15                       edgeToDelete ← (u, v)
16           deleteEdge(edgeToDelete)
17   if {cycles} ≠ ∅ then
18       removeCycles (V[G], E[G], s)
Algorithm findCycles (V[G], E[G], s)
 1   for each vertex u ∈ V[G] do          /* initialize for depth-first search */
 2       color[u] ← WHITE
 3       π[u] ← NIL                       /* used in Algorithm DFS-Visit */
 4   time ← 0                             /* used in Algorithm DFS-Visit */
 5   cycles ← { }                         /* initialize cycles to the empty set */
 6   DFS-Visit(s)                         /* depth-first search from source */
 7   for each edge (u, v) ∈ E[G] do       /* examine each edge in turn */
 8       if color[(u,v)] = GRAY then      /* gray edges denote back edges */
 9           cycle ← traceCycle (u, v)
10           cycles ← cycles ∪ {cycle}
11   return(cycles)
```

Figure 6.8 Algorithm for removing cycles from the graph.

Algorithm *RemoveCycles* insures that the timing graph is acyclic. The algorithm is presented in Figure 6.8.The first step detects if cycles exist in the timing graph. Cycles in the graph are found by performing a depth-first search from the source in order to characterize the edges of the graph. Initially, all nodes are white. During the examination of the node's adjacency list, the node is colored gray. When all edges have been examined the node is colored black. For completeness, the depth-first search algorithm is shown in Figure 6.9 [45]. The depth first search classifies edges as it encounters them. Each edge

Algorithm DFS-Visit(u)

```
1   color[u] ← GRAY                    /* white vertex u has just been discovered */
2   startTime[u] ← time ← time + 1
3   for each v ∈ Adj[u]                /* explore edge (u,v)
4       color[(u, v)] ← color[v]       /* color the edge based on adjacent node color */
5       if color[v] = WHITE then
6           π[v] ← u                   /* set the predecessor field */
7           DFS-Visit(v)               /* continue probing deeper */
8    olor[u] ← BLACK                   /* blacken u; it is finished */
```

Figure 6.9 Depth-first search algorithm. Adapted from [45].

(u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (line 4):

1. White indicates a tree edge,

2. Gray indicates a back edge, and

3. Black indicates a forward or cross edge.

We utilize the following lemma characterizing directed acyclic graphs [45].

Lemma 6.1

A directed graph G is acyclic if and only if a depth-first search of G does not yield back edges.

Proof

$\Rightarrow$: (by contradiction) Suppose there is a back edge (u, v) . Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, there is a path from v to u in G, and the back edge (u, v) completes the cycle. ■

$\Leftarrow$: (by contradiction) Suppose G contains a cycle c. We need to show that a depth-first search of G yields a back edge. Let v be the first vertex to be found in c, and let (u, v) be the preceding edge in c. At the time $startTime[v]$, there is a path of white vertices from v to u. By the white-path theorem (given below), vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Theorem 6.1

In a depth-first forest of a directed or undirected graph G, vertex v is a descendant of vertex u if and only if at step 2 in Figure 6.9, vertex v can be reached from u along a path consisting of entirely white vertices.

Proof

See [45] for proof.

Observe that the gray vertices always form a linear chain of descendents corresponding to the current calling stack of DFS-Visit. Searching always emanates from the deepest gray vertex. An edge that reaches another gray vertex reaches an ancestor.

Returning to Figure 6.8, it now follows that cycles may be detected by searching for gray edges classified after depth-first search. Whenever a gray edge is found, the cycle can be traced by following the predecessor fields of a node until arriving back at the edge. Each cycle is added to the set of cycles in line 10.

After the set of cycles is determined, each cycle is examined in turn. In lines 3 through

7, *removeCycles* checks the special case of *wired* outputs[1] found on buses. Here, the cycle formed by output nodes on the same net may be broken by removing any edge that does not connect to the source or target. The more general case of a cycle formed by signal feedback is handled in lines 8 through 16. In this case, we search for an output pin connected to an input pin which has the largest jump in $startTime[u]$, the time the pin was first discovered. The $startTime[u]$ field is incremented upon entering the depth-first search routine. Nodes with consecutive start times are adjacent to another. Hence, a large jump in $startTime[u]$ between two nodes means a large distance (in terms of number of edges) between them. Since we are seeking to break any feedback loop, the largest jump between an output and an input pin is the correct edge to break the cycle. It is not sufficient to remove the gray edges, for these edges may not be output to input connections.

Since we are guaranteed to obtain a directed acyclic graph, Dreyfus's method to find the *M* shortest paths may be used. Dreyfus's method finds paths in which repeated nodes are admissible paths in a general network. We seek paths without repeated nodes. Since the graph is a dag, repeated nodes are not possible. If repeated nodes were to exist, it would imply that the graph has a cycle, contradicting the assumption that the graph is acyclic.

The *M* shortest path algorithm is presented in Figure 6.10. It consists of two phases: shortest path determination and path augmentation. The shortest path calculation can be found in $O(V + E)$ time using the DAG-SHORTEST-PATH algorithm (phase *a*). This is optimal. In contrast, Dijkstra's method (needed for an undirected graph) requires (V^2) [50]. The second phase yields *M*-1 additional paths using Dreyfus's method. Dreyfus's method computes *M* shortest paths from the source to each of the other $n - 1$ nodes of the graph. Let d_u^m equal the length of the *m*th shortest path from the source to the node *u*. We define $n[u, v]$ to be the number of paths in which (u, v) is the final arc in the set of *m*

1. The set of output pins includes bidirectional and tri-state outputs.

Algorithm findMShortestPaths(s, t, M)

a ** First find the shortest path by using DAG-SHORTEST_PATH */*
1 topologically sort the vertices of G_d
2 **for** each vertex $v \in V[G]$ **do** */* initialize the graph */*
3 $d^1[v] \leftarrow \infty$ */* the distances are set to infinity */*
4 $\pi^1[v] \leftarrow$ NIL */* the predecessor is set to none */*
5 $d^1[s] \leftarrow 0$
6 **for** each vertex u taken in topological sorted order **do**
7 **for** each vertex $v \in$ Adj $[u]$ **do**
8 Relax(u,v,w)
b ** Find the M other shortest paths in O(Mnlogn) running time using Dreyfus's Method */*
9 Dreyfus_mpaths($V[G_d]$, $E[V_d]$, M, s, t)

Figure 6.10 Algorithm for find the M shortest paths in an acyclic graph. Phase *a* finds the shortest path in the dag while phase *b* augments the number of paths using Dreyfus's method.

shortest paths from the source to node v. The (m+1)st shortest path from the source to node v has some final edge (u, v). The length of this path from the source to u must be $d_u^{n[u,v]+1}$. By minimizing over all possible choices of u, we obtain

$$d_v^{m+1} = \min_u \{ d_u^{n[u,v]+1} + w(u, v) \} \tag{6.24}$$

In general, d_u^{m+1} appears on the right hand side of Equation 6.24 only if the edge (u, v) is the final edge in each of the 1st, 2nd,..., mth shortest paths from the source to node v. Since (u, v) is the final edge of the shortest path, it follows that the number of edges in a shortest path to node u is one less than the number of edges in a shortest path to v. Therefore, if the nodes are processed in the topological order, the value of d_u^{m+1} will be known when it is needed in the computation of d_v^{m+1}.

Dreyfus's method can be divided into two stages. Data structure initialization is performed in the first stage. Each node has a priority queue which stores the incoming edges classified by cost. In line 9, the distance is tested to see if it may enter the priority queue. If the path has not yet been processed, its entry into the priority queue will be delayed until

Algorithm Dreyfus_mpaths($V[G]$, $E[G]$, M, s, t)

```
a   /* initialize data structures */
1   for u ∈ V[G] do
2       edgeHeap[v] ← buildPriorityQueue()
3       futureEdge[v] ← { }
4       for m ← 2 to M do d_u^m ← ∞
5       for v ∈ Adj[u] do
6           if π[v] = u then
7               n[u, v] ← 1
8           else n[u, v] ← 0
9           if d_u^{n[u,v]+1} = ∞ then
10              Enqueue(futureEdge[v],d_u^{n[u,v]+1})   /* delay added to heap until cost is known */
11          else
12              insertPriorityQueue(edgeHeap[v],d_u^{n[u,v]+1})
b   /* done initialization now perform longest path search */
14  for m ← 2 to M do          /* M is the number of longest paths desired */
15      for each vertex v taken in topological sorted order do
16          while futureEdge[v] ≠ { } do        /* costs for edge are now known */
17              d_u^{n[u,v]+1} ← head[futureEdge[v]]
18              insertPriorityQueue(edgeHeap[v],d_u^{n[u,v]+1})
19              Dequeue(futureEdge[v],u)
20          if edgeHeap[v] ≠ { } then
21              d_u^{n[u,v]+1} ← ExtractMin(edgeHeap[v])
22              Relax-M(u,v,w)
23              n[u, v] ← n[u, v] + 1            /* increment number of paths for this edge */
24              if d_u^{n[u,v]+1} = ∞ then       /* delay is added to heap if necessary */
25                  Enqueue(futureEdge[v],d_u^{n[u,v]+1})
26              else
27                  insertPriorityQueue(edgeHeap[v],d_u^{n[u,v]+1})
28      path_m ← Trace-Back-From-Target(t)
```

Figure 6.11 Dreyfus's *M* shortest path algorithm. Phase *a* initializes the data structures. Phase *b* finds the *M* shortest paths from the origin. Edges may be put in the priority heap once the cost is known.

the distance is known. The edge's entry into the priority queue is delayed (and added to the *futureEdge* queue) if

$$n[u, v] + 1 = m + 1 \tag{6.25}$$

since the (m+1)st path has yet to be calculated.

The second stage (lines 14-28) constitutes the core of Dreyfus's algorithm. The nested loops of lines 14 and 15, in conjunction with the Extract-Min operation of line 21, combine to form the upper bound for the time complexity $O(M \cdot n \cdot \log n)$. Lines 16-19 add edges which have been delayed to the priority queue. After the first iteration of the loop, only one edge will be updated in lines 24-27. Finally, the mth shortest path is determined by tracing back through the predecessor fields. The predecessor fields are assigned in the relaxation routines shown in Figure 6.12. Observe that only one relaxation is done for each node selected in line 15.

Algorithm Relax(u,v,w)

```
1    if  d^1[v] > d^1[u] + w(u, v)  then
2        d^1[v] ← d^1[u] + w(u, v)
3        π^1[v] ← u
```

Algorithm Relax-M(u,v,w)

```
1    if d_u^n[u, v] + 1 ≠ ∞ then
2        if  d_v^m > d_u^n[u, v] + 1 + w(u, v)  then
3            d_v^m ← d_u^n[u, v] + 1 + w(u, v)
4            π^m[v] ← u
```

Figure 6.12 Relaxation routines.

6.3.6 Matched Critical Path Analysis Using Pin Pair Constraints

Matching pin-pair delays when using the pin-pair constraints is similar to the previous matched cases. Again, the user specifies a tolerance for the mismatch in pin-pair delays. In this case, we assign the penalty for two pin pairs p to be:

$$(p) = \begin{cases} match(p) - tolerance(p) & \text{if } match(p) > tolerance(p) \\ 0 & \text{otherwise} \end{cases} \quad (6.26)$$

where the match is now defined as difference in the arrival times,

$$match(p) = \max_{i,j} \left| T_a(p_i) - T_a(p_j) \right| \quad (6.27)$$

where the p_i and p_j are all the paths between the two pin pairs.

In this case, there are $O(M^2)$ constraints between the two sets of paths.

6.4 Results

The net length constraint algorithm has been used extensively within the industry. Both the Intel 486 and Pentium[1] microprocessors were designed using TimberWolfSC with net length constraints.

Figure 6.13 shows a small circuit (50 cells) in which net lengths have been constrained to a lower bound of 25 microns and an upper bound of 200 microns. This circuit is a sequential counter in which equalizing net lengths will increase the frequency of operation. Figure 6.13a shows the results of simulated annealing placement without net constraints. Most nets have small wire lengths but several have lengths over 200 microns. In Figure 6.13b, the distribution of nets converges to an average of 100 microns. Additionally, there were no nets shorter than 25 microns. Several net lengths have remained unchanged; these are the power and ground nets which connect to every cell. These net constraints were entered to show that the algorithm is tolerant to infeasible constraints.

1. Pentium is a registered trademark of the Intel Corporation.

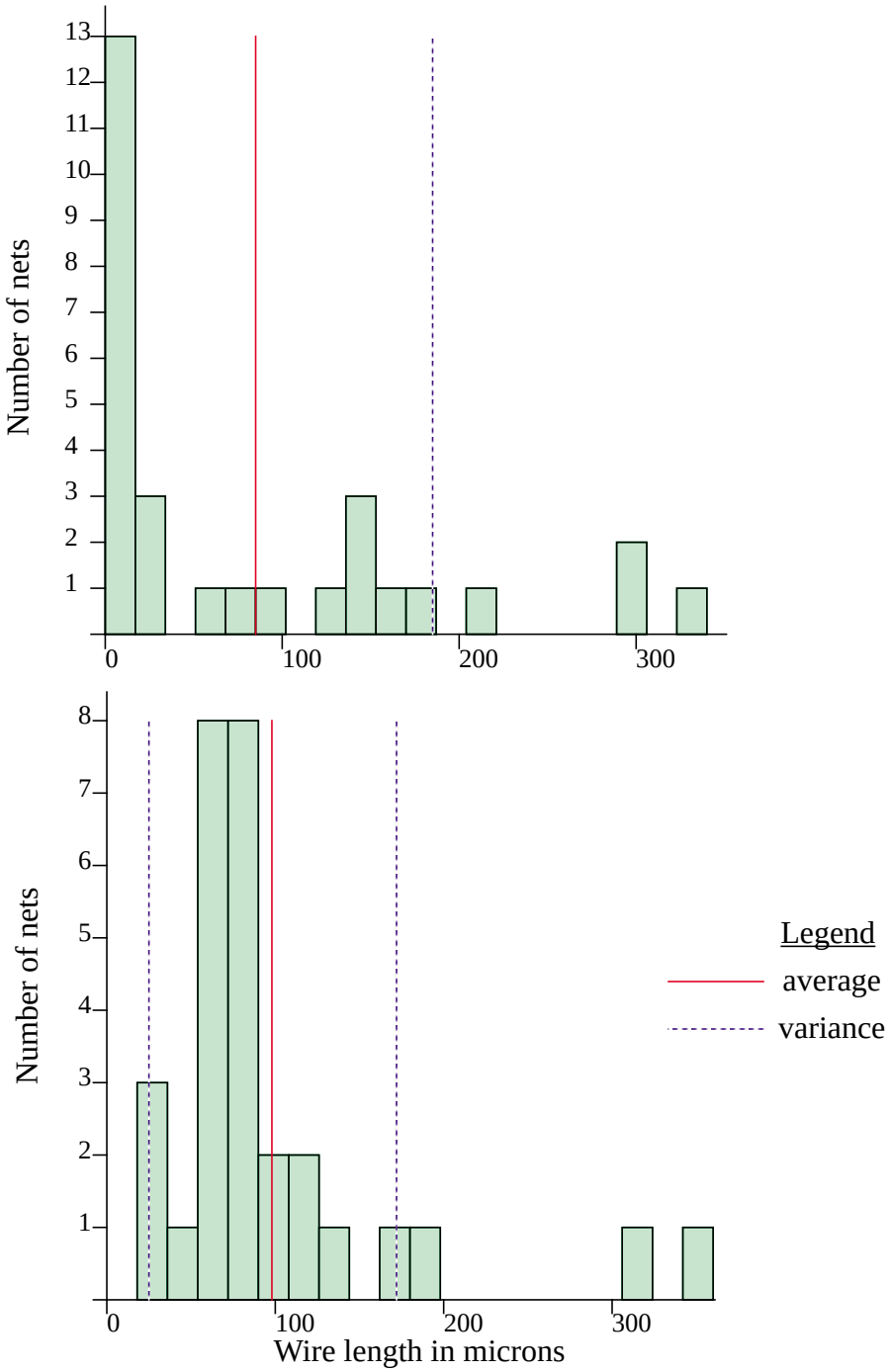

Figure 6.13 Net distribution for small circuit. Figure *a* is placement without timing constraints. Figure *b* is constrained placement.

Table 6.1 presents all of the critical paths for the MCNC benchmark circuit *fract*. These paths were automatically detected using the pin-pair timing constraint algorithm; user interaction was not required. Because it was a synchronous output, combinatorial paths were not found to primary output pin *W.*

Table 6.1 Number of timing critical paths for MCNC benchmark circuit *fract.*

primary input	Number of paths to primary output Z	Number of paths to primary output W
reset	-	-
Phi1H	-	-
Phi2H	-	-
X	17	-
Clear	-	-
CD16	1	-
CD15	1	-
CD14	1	-
CD13	1	-
CD12	1	-
CD11	1	-
CD10	1	-
CD9	1	-
CD8	1	-
CD7	1	-
CD6	1	-
CD5	1	-
CD4	1	-
CD3	1	-
CD2	1	-
CD1	1	-

Table 6.2 Time for longest path for pin pair X to Z in nanoseconds ($K = \infty$) **.**

run	no constraints	M=1	M=5	M=17
1	198	130	131	129
2	209	134	125	134
3	211	135	119	133
4	214	136	126	124
5	209	137	119	129
average	208	134	124	130

The most interesting pair of I/O pins are *X* and *Z* with 17 unique paths between them. Table 6.2 compares the effect of varying *M*, the number of paths monitored, for the longest path in the circuit. The results using the pin-pair timing algorithm are dramatic; the circuit runs at least 34% faster! In monitoring the 17 paths, it was discovered that five paths were much longer than the others. By monitoring just these five paths, the best results were obtained. In this case, the longest timing path through the chip was reduced by more than 40%! This is because the algorithm was not distracted by the easily satisfiable paths.

A decrease in the time delay for the other paths was also realized. Figure 6.14 displays the timing delay distribution. Notice that the average time delay in the circuit was reduced from 115 nanoseconds to 37 nanoseconds, a savings of 67%.

Table 6.3 Wire length and run time results for *fract* circuit ($K = \infty$) **.**

	no constraints	M=1	M=5	M=17
wire length	57276	60691	63073	61209
area (μm^2)	1.56×10^6	1.60×10^6	1.68×10^6	1.58×10^6
run time (secs.)	670	801	966	1201

Table 6.3 compares wire length, area after global routing, and execution times for each case. When *K* is set to infinity and *M* is set to 1, the integrated circuit is 34% faster, with an

area penalty of only 2.5% and an execution time increase of 16%. Generally, the area after global routing correlates with the wire length after placement. For this case, dynamically searching for the longest of the set of paths yields excellent results with minimal increase in the execution time.

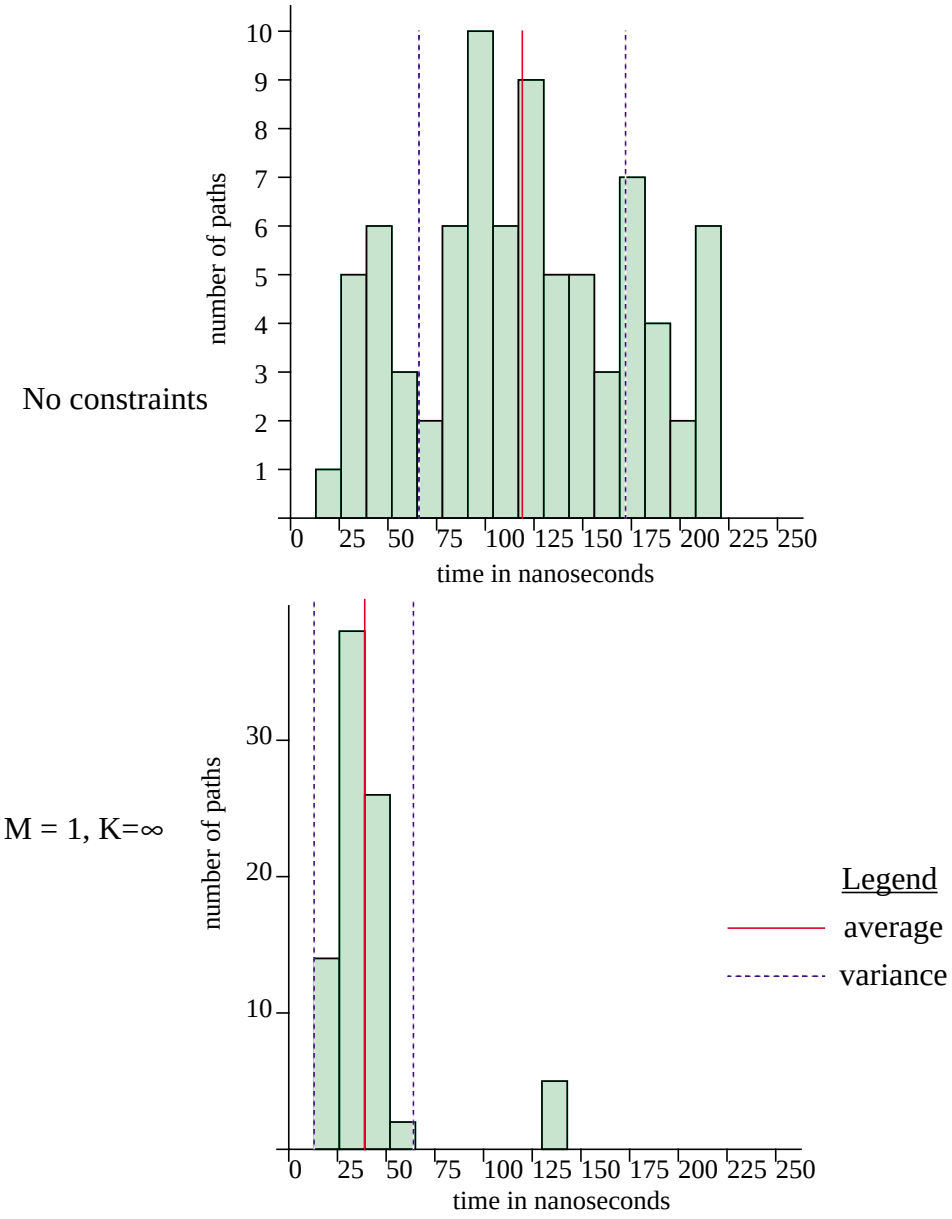

Figure 6.14 Dramatic improvement in timing path distribution. The graphs plot number of paths at a given time delay.

To further reduce the run time, the number of time critical paths was pruned to the 5 costliest paths. Table 6.4 shows the delay of the longest path using pruning. Notice that the longest delay path did not not change. In Table 6.5, we see that the area after global routing also remained the same. In addition, the cpu time was reduced.

Table 6.4 Time for longest path for pin pair X to Z in nanoseconds using pruning.

run	M=1, K=∞	M=1, K=5
1	130	134
2	134	133
3	135	130
4	136	127
5	137	130
average	134	131

Table 6.5 Wire length and run time results for *fract* circuit using pruning.

	M=1	M=1, K=5
wire length	60691	60322
area (μm^2)	$.60\times10^6$	$.60\times10^6$
run time (secs.)	801	750

Figure 6.15 shows the delay distribution for the *struct* benchmark. Notice that the average delay in the circuit was reduced from 320 nanoseconds to 160 nanoseconds, a savings of 100%. In all cases, the user needs no knowledge of the time critical paths of the circuit.

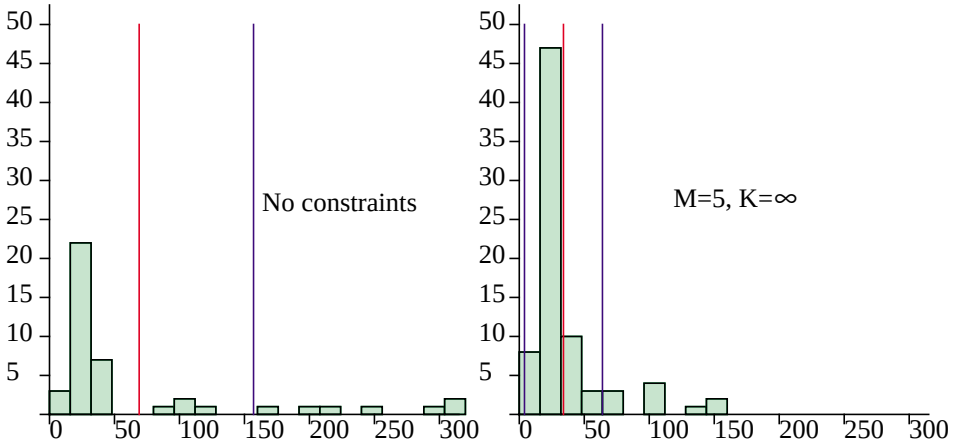

Figure 6.15 Timing results for the *struct* circuit.

6.5 Conclusions

We have presented six algorithms for controlling time delays in an integrated circuit. A novel pin-pair algorithm controls the delay without the need for user path specification. The algorithm has optimal time complexity. Using this algorithm, we presented results for the MCNC benchmark circuit *fract*. This is the first report of timing driven placement results for any benchmark circuit. The algorithm was able to increase the speed of the chip by 34% at an area cost of only 2.5%.