INTERCONNECT DELAY

As feature sizes continue to shrink and chip area becomes larger in integrated circuits, the importance of interconnect delay increases rapidly with respect to gate delay. As a result, interconnect delay at the global level has become a critical factor in determining the system performance in deep submicron designs. In this chapter, we will address the interconnect delay issue in balanced buffering problems in order to obtain more realistic buffer tree solutions, in terms of delay and topology. Our emphasis is to devise a buffer tree construction method based on the mathematical and algorithmic concepts provided so far. In order to reach an effective buffering method, we will take into consideration the wire's physical properties, such as wire resistance and wire capacitance, as well as all buffering issues discussed thus far, including optimal tree topology, buffer library handling and phase shifting in the presence of inverting buffers. Note that we do not address the wire inductance in this work.

Although the general buffering problem is NP-Complete[1], we have shown that balanced buffering can be solved with a reasonable amount of memory and runtime. We now introduce a new version of our algorithm where interconnect delay is considered. While the general problem solving approach is still based on the branch-and-bound algorithm, we will yet again profit from Dynamic Programming to efficiently build up our algorithm and partially resolve the time complexity issue. However, due to the limited time in our Master's project, the new ideal delay required for the bound calculation is not provided in this dissertation and is left for prospective extensions on this work. Instead, we have roughly modified the old bound used to avoid an exhaustive solution enumeration, and only concentrate on proposing a well-structured method to generalize the buffering algorithm previously introduced. In order to establish a firm basis for our algorithmic structure, we will simplify some design problems whenever appropriate and in a manner that the method generality and the solution quality are not hurt.

Chapter Outline

An efficient modeling system is required in order to properly handle the interconnect issue. Section 4.1 provides a suitable model to take into account the physical properties of buffer tree

[1] As mentioned in the first chapter, a simultaneously buffer insertion and optimal topology search has been proven to be a NP-Complete problem by Berman [BERMAN 89].

topology. Section 4.2 formulates the buffering problem with interconnect delay considerations. Section 4.3 partially modifies the bound required by the Branch-and-Bound algorithm. As the buffer tree topology is now sensitive to wire delay, the number of possible buffering solutions in the search space increases. Section 4.4 explains how a comprehensive search space covering all possible buffering solutions can be created to guarantee the solution quality of the buffering algorithm. In section 4.5 we review the structure of the buffering algorithm under new conditions. Section 4.6 introduces a local delay optimization technique that partially improves the solutions delay before adding them to the search space. Section 4.7 and 4.8 discuss the approaches by which one can profit from speedup techniques provided in chapter 3. The final buffering algorithm with interconnect delay taken into account is given in section 4.9. In the end, section 4.10 examines the runtime of the new buffering algorithm.

1.1 How to Model the Problem

1.1.1 Wire Segmenting

Previously we assumed at most one buffer of a given type to be placed on a single wire in a balanced buffer tree. This constraint can severely hurt solution quality if the wire delay is taken into account, as multiple buffers are required to effectively drive wires with large lumped RCs. To circumvent this problem, one can divide each wire into multiple smaller *segments*, as illustrated in Figure 4-1.

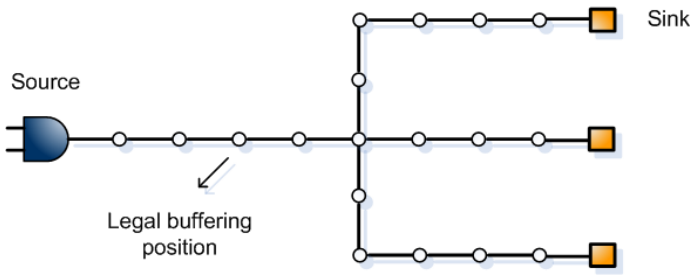


Figure 0-1wire segmenting

By wire segmenting, we actually introduce new legal buffering positions to insert buffers. Although segmenting each wire into small wires can help finding better buffering solutions, establishing the right number of segments in which a certain wire must be divided is crucial. A small number of wire segments may result in sub-optimal solution, whereas a large number of wire segments may significantly increase CPU time.

The ideal number of wire segments has been studied by [ALPERT 97]. The authors compute the appropriate number of wire segments using only one buffer type. They also suggest handling the buffer library by obtaining the ideal wire segmenting factor for each buffer type, and choosing the maximum number of wire segments achieved for different buffers to guarantee the solution optimality.

We do not use the work [ALPERT 97] to find the best segmenting factor for a wire. Instead, we propose the minimum number of wire segments needed to model the balanced buffering problem, and suggest using the ideal number of wire segments presented by [ALPERT 97] for further extensions. Since the way of dividing a wire does not affect the algorithm or the assumptions on which our delay calculations are done, we begin with the most simplistic form of dividing the buffer tree wires into smaller segments.

1.1.2 The Grid Graph

An effective approach to model the balanced buffering with segmented wires is to fit the net into a grid graph. The grid must be a unit distance graph[1] as the basic length of a wire segment is equal everywhere in the buffer tree, and is defined by the intersection of horizontal and vertical lines running through the terminals of the net.

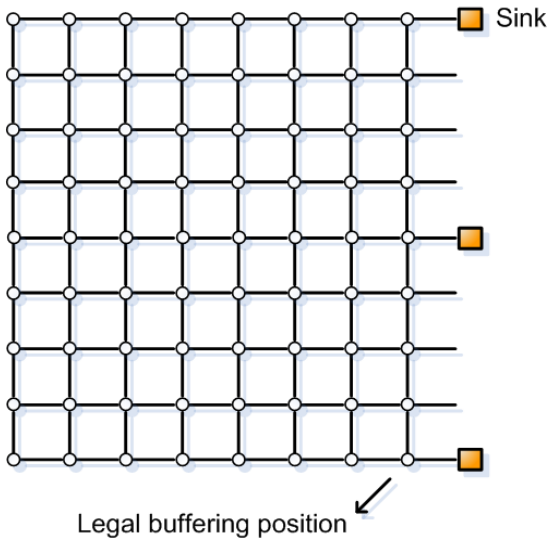


Figure 0-2 A typical grid

The edges of each square block in the grid provide the legal routing paths, while the angles of each square represent the legal buffering positions. Thus, having **m** vertical and **n** horizontal

[1] A unit distance graph is a graph formed from a collection of points in the Euclidean plane by connecting two points by an edge whenever the distance between the two points is exactly one. [ERDOS 46]

lines, the ultimate grid into which a buffer tree is produced has $(m-1)(n-1)$ squares, and **mn** legal buffering places. The vertical length of the grid must exactly correspond to the distance between the first and the last sinks (assuming the sinks are vertically ordered in a column), whereas the horizontal length of the grid is variable and is dependent upon the number of stages of the buffer tree. We will discuss more about the grid horizontal length later on.

As each edge in the grid graph represents a wire segment, we can model the distributed RC net of a buffer tree as illustrated in Figure 4-3, and use it in the Elmore delay formula introduced before.

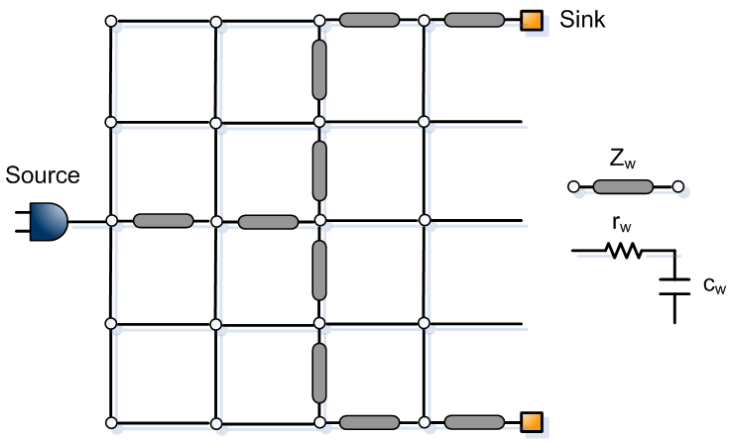


Figure 0-3 Distributed RC modeling

1.1.3 Efficient Topology Modeling

Having wire delay taken into account, the buffer tree delay is now sensitive to topology. Therefore, one has to carefully choose the appropriate topology, based on how we search the best buffer tree. As we consider the wire delay in our delay calculations, we are willing to minimize the total vertical and horizontal wire lengths of the generated tree, hoping that it would also minimize the total wire delay. To this end, we can picture a tree with the least wire-overlap. For a balanced buffer tree, where all the sinks are vertically placed in a column, the actual wire-overlap that affects the critical path delay occurs just for vertical wires, given the fact that there is no wire-overlap for horizontal wires on the critical path. In order to minimize the length of overlapped vertical wires, one would suggest the solution in Figure 4-4, where the intermediate buffers placed at the sub-tree roots are found as close as possible to the source gate:

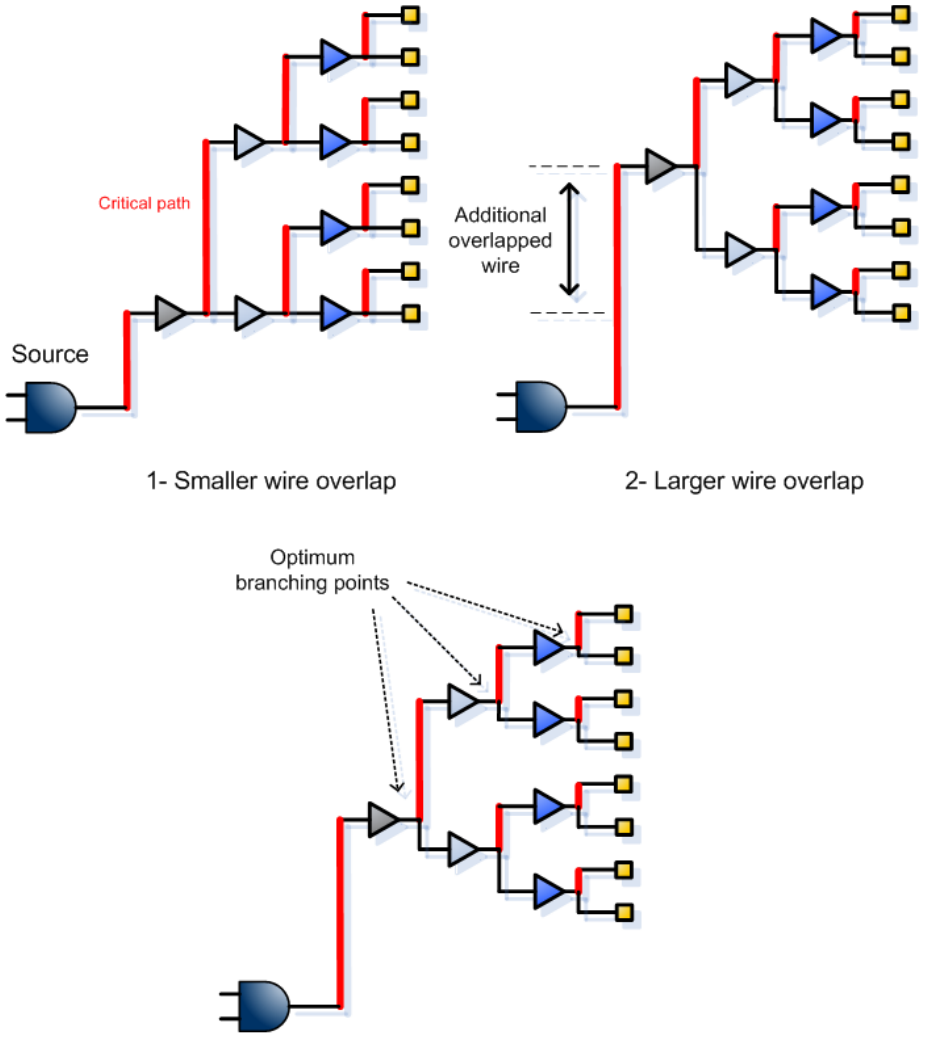


Figure 0-4 Three different vertical wire distribution

This effectively reduces the total length of vertical wires. Yet reducing wire-overlap does not necessarily guarantee a better delay, as the size of each single vertical wire appearing in the critical path is now maximized. As the wire delay increases quadratically with the wire length increase, the size of each single vertical wire becomes crucial. It can be shown that the optimum branching point, where the root of the subtree is defined, is always found somewhere between the beginning and the middle of the vertical wire. Using the optimum branching points all across the buffer tree results in an efficient distribution in terms of vertical wire lengths. It can also be shown that as the fanout number increases and larger vertical wire emerge, the optimum branching point moves toward the middle of the wire. In order to simplify the problem of finding such a branching point, we can decide whether to always use the middle or the beginning of a wire as the subtree root.

Very long vertical wires are regularly encountered in buffer tree construction for a source gate with large fanout numbers. Since we usually deal with large fanout numbers in practice, we select the middle of a vertical wire to insert the buffer root of the subtree. Such decision yields a relatively satisfying approximation of the optimum vertical wire distribution for large fanout trees. Yet, one could still suggest improving the buffer tree delay by minimizing the wire-overlap for small fanout numbers.

In order to implement that method, we need to consider at least one legal buffering position in between two adjacent sinks. As stated before, we study the most simplistic form of the buffering problem with wire delay considerations; therefore we choose maximum and minimum one graph node as the legal buffering position between two sinks, resulting in the following grid structure:

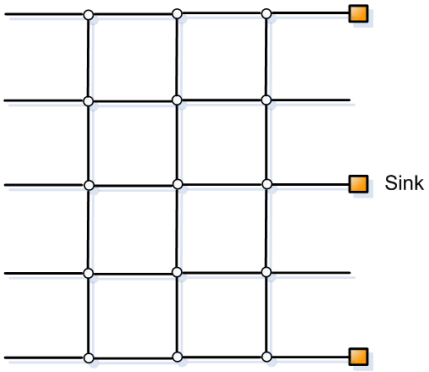


Figure 0-5 A grid with 2 squares between each pair of sinks

1.1.4 Simplifying the Problem

In order to limit the enormous search space of possible buffer trees, we ignore the horizontal wires in the buffering process. This simplification is done for two main reasons:

1- We do not need to have more than one horizontal wire segment at each buffering stage, assuming that there is no placement restriction posed by the problem. We can therefore modify the output resistance and the intrinsic delay of a buffer driving a single wire segment such that the same delay value is achieved.

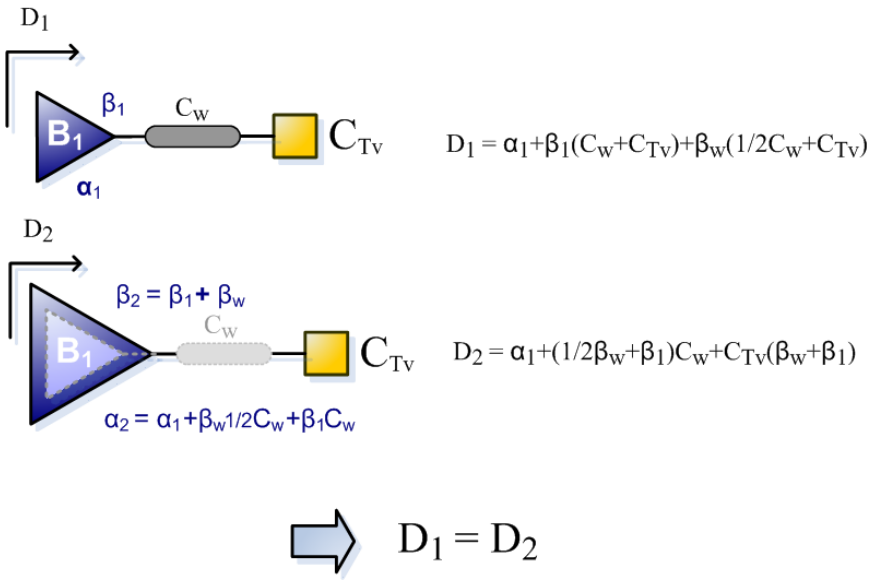


Figure 0-6 Ignoring horizontal wire delay by modifying the source buffer

2- The impact of horizontal wires on the total interconnect delay is dominated by that of vertical wires, particularly for large fanout numbers. Having n fanouts, the buffer tree has $\log_2 n$ stages in the worst case (and hence $\log_2 n$ horizontal wires participating in the critical path), whereas it always has $\frac{(n-1)}{2}$ vertical wire segments on the critical path.

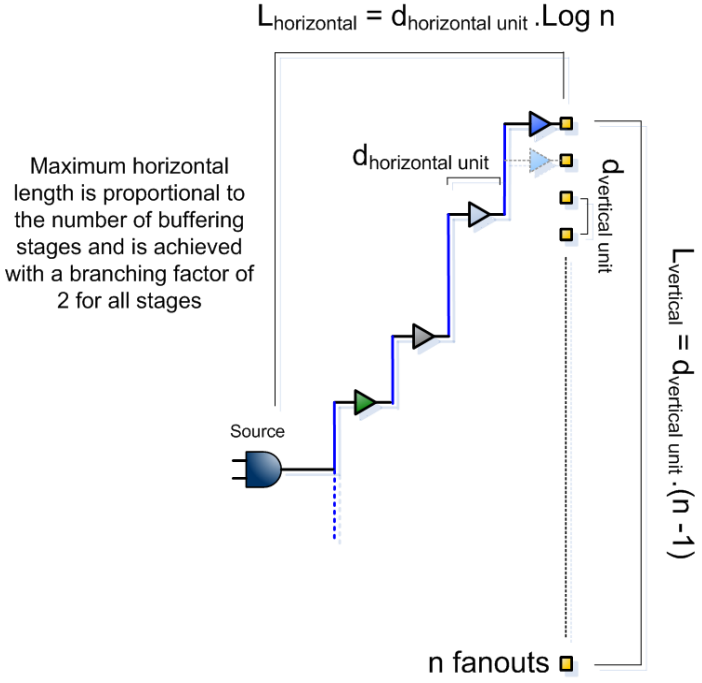


Figure 0-7 The impact of horizontal and vertical wires on the buffer tree delay

As a result, we will only concentrate on vertical wires in searching for the best buffer tree.

1.2 Problem Formulation

We assume that the topology is not fixed. In the grid graph, a routing tree $T = (V, E)$ consists of a set of *n* nodes *V* and a set of (*n-1*) wires *E*. The set *V* is the union of the unique source gate *G*, internal nodes *I* and the sink nodes *S*:

$$V = \{\{G\} \cup I \cup S\}$$

A wire $e \in E$ with length l_e is an ordered pair of nodes $e = (u, v)$ in which the signal propagates from *u* to *v*. Note that for each node $v \in I \cup S$ there is a unique *parent wire* $(u, v) \in E$.

In a buffer tree with segmented wires we have to deal with distributed RC trees. Following [OKAMOTO 96B] [LILLIS 95] we adopt the Elmore delay [ELMORE 47] for interconnect delays and a linear model for gate delays. Using a "*L-form*" RC model, the Elmore delay of a distributed RC tree is:

$$D_i = \sum R_{ik} \sum C_k$$

Where C_k is every capacitance in the network in sequence, and R_{ik} is common resistance in path between source and node *i* and source and node *k*. Since the size of each segmented wire is assumed to be equal everywhere in the buffer tree, we can rewrite the Elmore's equation with a predefined wire unit resistance R_{unit} and wire unit capacitance C_{unit}. Let T_v denote the subtree rooted at *v* and the *load* at node *v* be the total capacitance $C_{T(v)}$. If $T(v) = (\{\{v\} \cup I' \cup S'\}, E')$ then:

$$C_{T(v)} = \sum_{u \in I' \cup S'} C_u + \sum_{e \in E'} C_e$$

The delay for a wire with *n* sections and a load capacitance $C_{T(v)}$ is:

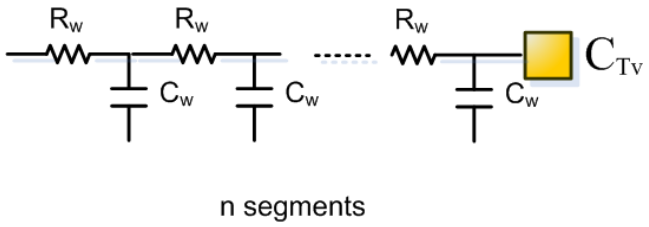


Figure 0-8 a segmented wire with a load capacitance $C_{T(v)}$

$$D_{wire} = \left(R_{unit}C_{unit} + 2R_{unit}C_{unit} + 3R_{unit}C_{unit} + ... + n.R_{unit}C_{unit}\right) + n.R_{unit}C_{T(v)}$$

$$D_{wire} = \frac{n(n+1)}{2}R_{unit}C_{unit} + n.R_{unit}C_{T(v)}$$

Where R_{unit} and C_{unit} are the respectively wire unit resistance and the wire unit capacitance. Given that:

$$n.R_{unit} = R_{wire} \text{ and } n.C_{unit} = C_{wire}$$

The delay formula becomes:

$$D_{wire} = R_{wire}\left(\frac{C_{wire}}{2} + \frac{C_{unit}}{2} + C_{T(v)}\right)$$

If the wire is long enough, we can ignore $\frac{C_{unit}}{2}$ and get the Elmore delay:

$$D_{wire} = R_{wire}\left(\frac{C_{wire}}{2} + C_{T(v)}\right)$$

The lumped RC model preserves the property that the Elmore delay is the same for a given wire no matter how the wire may be subdivided. The delay through a gate at node *v* is the same linear function that we introduced in the first chapter:

$$Delay_{gate\,v} = \alpha_v + \beta_v C_{T(v)}$$

Where α_v is the intrinsic delay and β_v is the output resistance of the buffer. In our delay calculations, we combine the Elmore delay models for RC trees and that for buffer gates. The delay from a node $v \in V$ to a sink *s* is:

$$Delay\,(v \to s) = \sum\left(Delay_{wire:(u,w)\in path(v,s)} + Delay_{gate:(u)}\right)$$

Where $path(v,s)$ is the set of edges on the path from *v* to *s*, (u,w) is the current edge and *u* is the legal buffering node in the grid graph. If *u* is not a gate but simply an internal node, then $Delay_{gate:(u)} = 0$. Note that off-path buffers decouple the load and effectively serve as "sinks" of T_v, while buffers on the path from *v* to *s* serve as internal nodes with non-zero delays.

We are given a buffer library which consists of inverting and non-inverting buffers. The loads are all identical in terms of capacitance and required time. No buffer sizing is applied to reduce the circuit delay. No phase-shifting is allowed to occur in the generated buffer tree in the presence of inverting buffers.

Balanced Buffering Problem: Given a set of identical sinks *S*, a buffer library *B* and a source gate *G* find the best buffer tree such that:

$$D_N: \textit{Minimized}$$

Where D_N denotes the overall delay of the net representing the balanced buffer tree.

1.3 The Bound

We do not calculate a new ideal delay formula with interconnect delay considerations, as the emphasis is on method adaptation. However, we need to have a bound to implement the branch-and-bound algorithm. If we utilize the previously calculated bound for balanced buffering, it would be too small as it does not take into account the physical wire properties. We present a safe modification on the ideal delay equation in order to have somewhat more realistic delay values.

If we follow the critical path in a buffer tree, we can extract the total wire capacitance encountered along the path that has to be driven by the best buffer in the ideal buffer tree. To that effect, we isolate the wires appearing in the critical path and we disregard the remaining wires of the buffer tree. The delay path is then a step-wise wire with additional vertical wires at each branching point, as shown in Figure 4-9. We have already given the constraint of placing the root buffers exactly in the middle of a vertical wire. Therefore, at each branching point we have a vertical wire equally divided by two.

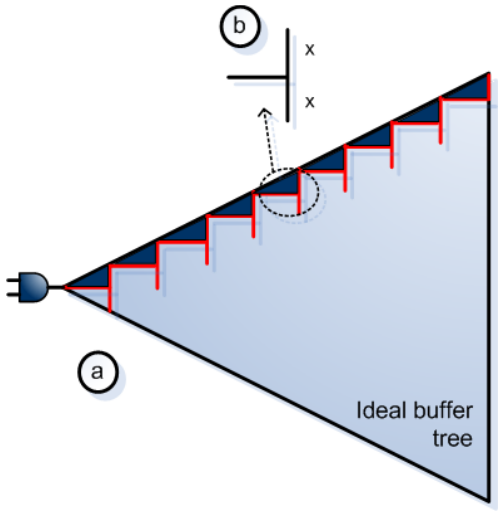


Figure 0-9 a- Isolated critical path
b- Equally divided vertical wire

By detaching the lower half of the vertical wires and placing them on the bottom side of the buffer tree, we obtain:

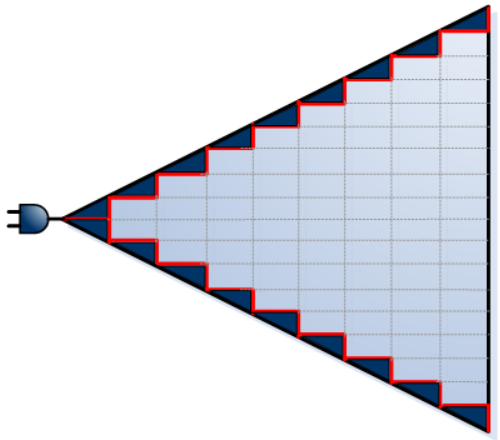


Figure 0-10 Mirrored critical path

Observe that we also mirror the horizontal wires in order to duplicate the critical path on the bottom side of the buffer tree. Yet, as we will see, the additional horizontal wires do not affect our bound calculation. Since the ideal balanced buffer tree is always an isosceles triangle, we obtain an identical form for the mirrored wires. Now the total length for vertical and horizontal wires is simply found by reflecting each sides of the triangle on its height and base. As a consequence, the total length of the vertical wires is found as the size of the triangle base, whereas the total length of the horizontal wires is found as the size of the triangle height (Figure 4-11 a).

Provided that we supposed the interconnect delay of the horizontal wires to be negligible during the buffer tree construction, and given that all wires in the ideal buffer tree are driven by the best buffer (except for the first stage), a part of the total interconnect delay is calculated as:

$$D_{VerticalWire} = \beta_{BestBuffer}.C_{Height}$$

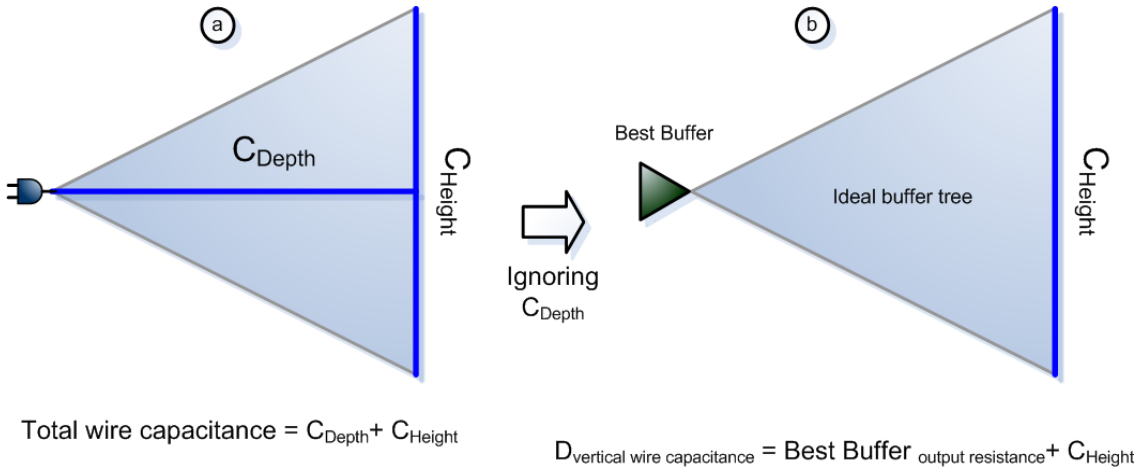


Figure 0-11 a- Obtaining Total wire capacitance
b- The best buffer driving the total vertical wire capacitance

Therefore the new ideal delay of the buffer tree is:

$$D_{optimum} = \mu\left(1+.\ln\left(\frac{\beta_0\gamma_T}{\mu}\right)\right)+D_{\text{VerticalWireCapacitance}}$$

Where

$$\mu = \beta_b\gamma_b e^{\left(\frac{\alpha_b}{\mu}+1\right)}$$

Having the sink positions, one can easily calculate the base of the triangle. As the sink positions are fixed and independent from the buffer tree structure, the additional delay is always extracted without any knowledge about the ideal balanced buffer tree properties. Although one can argue that taking the interconnect delay into account in the ideal delay calculation could result in different best buffers from the one found by interconnect free assumptions, our tests have shown the safety of the proposed modification.

1.4 Optimal Structure

In order to guarantee a good solution quality with a branch-and-bound algorithm, we have to enumerate practically all possible buffer trees. One necessary means to achieve this goal is to use a well-defined recursive function to generate the subproblems (as we did in the first chapter). We are dealing with a much more complex feasible region as the buffers can also be placed on the vertical wires. We have identified an efficient method with which we are able to recursively divide a buffering problem into smaller buffering subproblems. The ultimate recursive function responsible for building up the feasible region consists of 3 different smaller recursive functions. Before introducing those 3 types of recursive functions, we examine them in an example.

Consider the following example of an arbitrary best buffer tree:

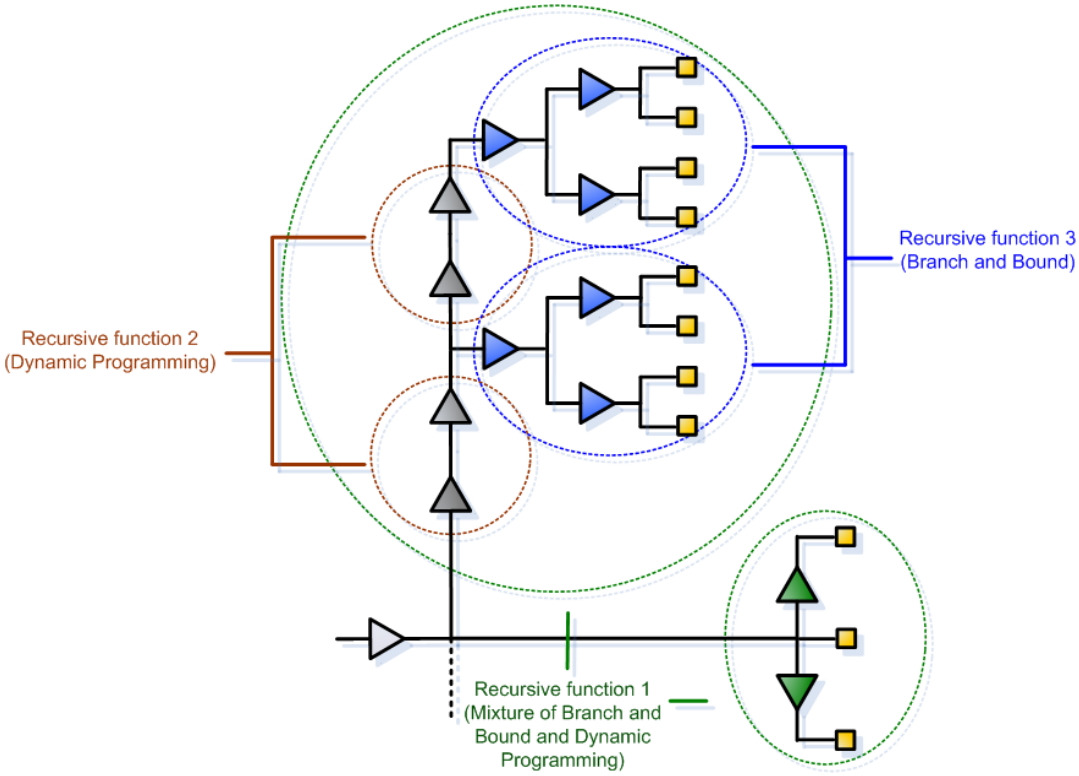


Figure 0-12 Subproblem generation

Concentrating only on the branching points of the buffer tree, we identify a main branching point right after the source gate and several local branching points over the rest of the buffer tree. While each buffer subtree produced by the main branching point has a different structure from its counterpart, it presents a homogenous buffering structure inside, where all its sub-sub-trees have the same structure. Therefore the main branching point forms two different subproblems. This

kind of subproblem generation is achieved through *Wire Decoupling* and the subtrees produced are called *Decoupled Subtrees*. The second type of subproblems is related to those buffer trees which share the same branching factor. They are categorized as balanced subtrees for which a recursive function was introduced in chapter 1. The third group of subproblems is a collection of buffered vertical wires. We will present the recursive structure of this group in section 4.6. The combination of the 3 subproblem classes forms a complete feasible region with all possible buffer trees. We summarize them as follows and will study their correlations in the following sections:

1- **Decoupled Subtrees**: those subtrees with dissimilar fanout arrangements
2- **Balanced Subtrees**: those subtrees with similar fanout arrangements (explained in chapter 1)
3- **Buffered Vertical Wires**: single vertical wires on which we perform buffer insertion

We now explain the two new recursive structures, *Decoupled Subtrees* and *Buffered Vertical Wires*.

1.4.1 Wire Decoupling

Wire decoupling is one of the three essential subproblems that guarantee a full set of possible buffer trees. The purpose of this section is to show how and under which conditions wire decoupling is performed. The usage of wire decoupling will be studied in section 4.5.

Given that the interconnect delay is now considered in our delay calculations, different fanout configurations can result in different solutions, as the arrival time of each sink is now sensitive to topology. For more convenience, and to preserve the symmetrical structure of the balanced buffering problem, we pair the sinks with identical distance from the source gate and use those pairs in the *Wire Decoupling* function.

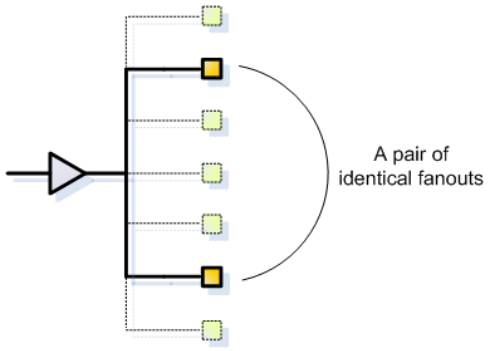


Figure 0-13 a pair of identical fanouts

We first provide the necessary assumptions based on which the wire decoupling function is defined. These assumptions are based on the method we used to model different fanout arrangements.

Fanout Encoding

We have to deal with a specific type of subtrees called “Decoupled Subtree”. When a group of fanout pairs are extracted from a tree, the tree is said to be *decoupled*. The two subtrees created by such an action are called *Left Decoupled Subtree* and *Right Decoupled Subtree*.

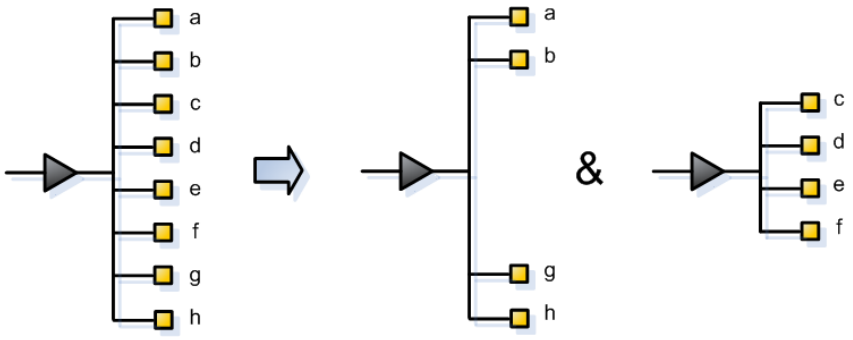


Figure 0-14 a tree is decoupled into two subtrees

One major issue in dealing with this type of subtrees is the existence of two subtrees with the same fanout number but with different sink arrangements (Figure 4-15). As they might result in different best buffer trees, we must differentiate them. A tree, whether decoupled or not, can be encoded with 3 parameters:

1) The axis of symmetry (as we deal with symmetrically balanced trees)

2) The Fanout number

3) The vertical distance (in terms of grid square block) between the axis of symmetry and the nearest sink, called Δ.

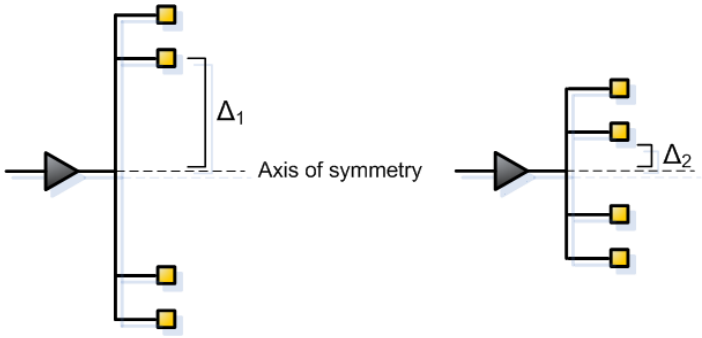


Figure 0-15 two trees with a same fanout number and different Δ

Having these parameters, we are able to uniquely encode a tree everywhere on a grid graph. Among those three parameters, Δ is the most important one as it reflects the fanout arrangement of a tree. We will see the application of Δ in further simplifying the problem definition. On a grid graph with a distance of two square blocks between each couple of adjacent sinks[1], if $\Delta=0$ or $\Delta=1$ we call the given tree a *uniform* tree, while for $\Delta>1$ we have *non-uniform* trees.

$$0 \leq \Delta \leq 1 \qquad \text{Uniform Tree}$$
$$1 < \Delta \qquad \text{Non-Uniform Tree}$$

The main reason to categorize the trees into uniform and non-uniform types is to determine which subtree is a left decoupled subtree and which one is a right decoupled subtree. A left decoupled subtree has always $1 < \Delta$ and is non-uniform as it is the remnant of an original uniform tree from which a group of fanout pairs have been extracted. As we will see, we forbid any more wire decoupling for non-uniform subtrees in order to preserve the consistency of the main algorithm.

Based on the assumptions presented for fanout encoding, we now introduce the function which is responsible for decoupling the trees.

Wire Decoupling Function

Given a group of fanouts with known positions, our objective in wire decoupling is to create a set of possible combinations of the fanout pairs. We are only interested in *legal* subtrees, which means the subtrees that have a group of contiguous fanout pairs. We will later explain that this constraint eliminates any possible conflicts in combining *Wire Decoupling* with the other two subproblem generating techniques. As an example, for a tree with a fanout number of 4, one illegal subtree along with the possible legal subtrees expected from a wire decoupling process would be:

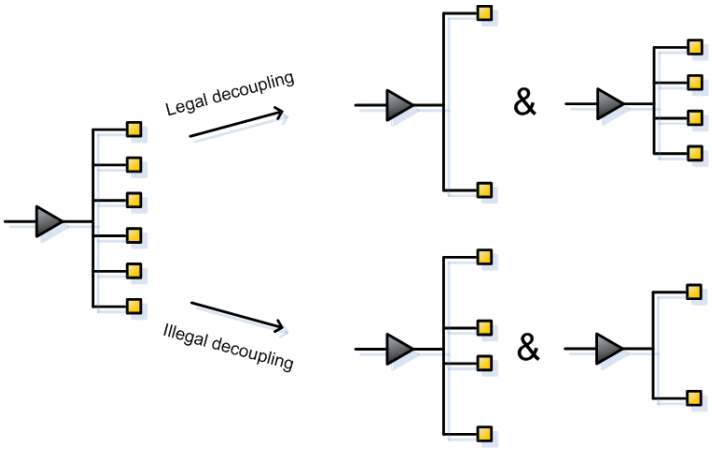


Figure 0-16 Legal and illegal tree decoupling

[1] That simplification has been used throughout this master's project. Other grid resolutions require reusing the underlying concepts.

To obtain the full set of legal decoupled subtrees, we define a recursive function as follows:

Wire Decoupling (a uniform tree)
{
    Save the current tree configuration in *Solution Set*
    Define a boundary and make it point to the first fanout pair

    While (there is any fanout pair after the boundary)
    {
        Temporary Solution Set = **Wire Decoupling** (right decoupled subtree)
        For (the number of solutions in *Temporary Solution Set*)
        {
            Connect the decoupled sub-subtrees of *Temporary Solution Set* to the current left subtree
            Save the resulting decoupled tree in *Solution Set*
        }
    }
    Return *Solution Set*
}

Figure 0-17 Wire decoupling algorithm

In order to explain the functionality of the Wire Decoupling function, let us find all legal decoupled subtrees for the given tree:

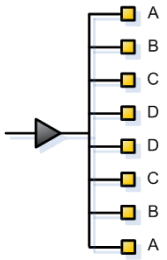


Figure 0-18 The decoupling example

We define a separating line as the *boundary* and place it after the first fanout pair **A**. As we now have two subtrees, **A** and **BCD**, we save the decoupled tree as a solution. Subsequently we find all possible solutions to **BCD** in the first while loop and combine them with **A**, the result of which is a set of multiple decoupled subtrees. Moving the boundary one fanout pair forward, we obtain a new solution with **AB** and **CD**. Repeatedly moving the boundary forward and recursively solving the right subtrees, a full set of all possible subtrees is achieved.

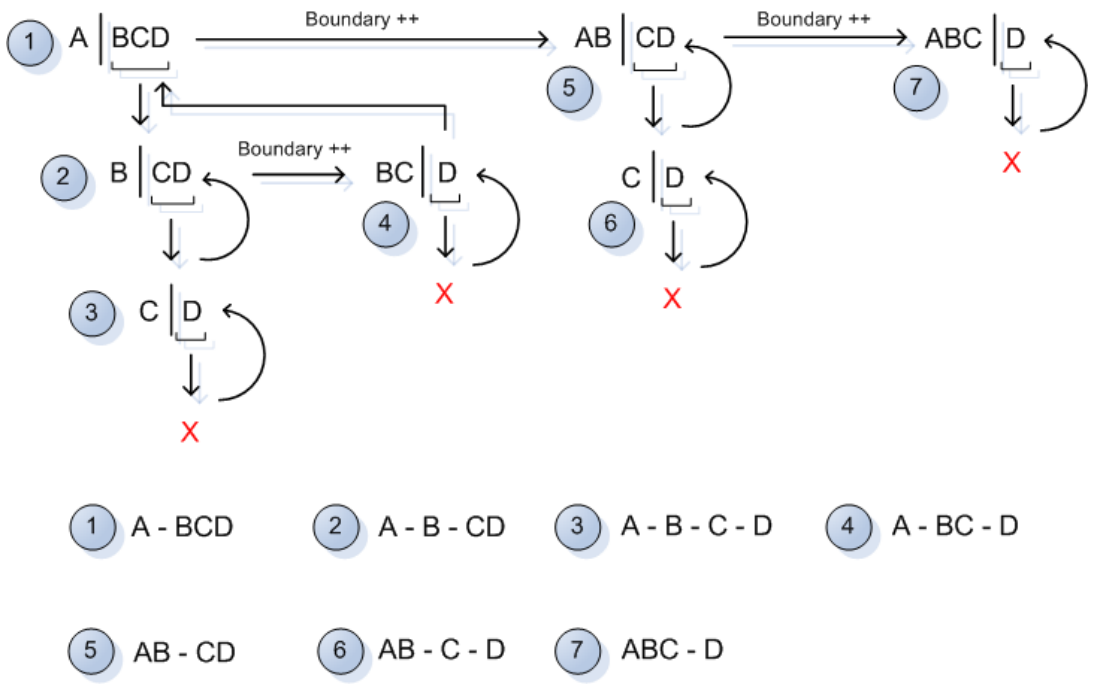


Figure 0-19 Generating legal decoupled trees for a tree with 4 fanout pairs

The application of the recursive structure of wire decoupling and the way it contributes to the search procedure will be studied in section 4.5.

Properties of the Wire Decoupling Function

Our purpose of moving the boundary in a left-to-right manner (or in fact from the most external fanout pair to the most internal one) in the wire decoupling function is to only decouple the uniform subtrees with $0 \le \Delta \le 1$. This property makes wire decoupling consistent when we combine it with two other subproblem generating methods in the main algorithm.

Brief Introduction to Vertical Buffering

To efficiently deal with the huge solution space, we insert the buffers on the vertical wires in an independent procedure, called *Vertical Buffering*. For every decoupling solution (used in wire decoupling) or dividing solution (used in producing balanced subtrees) we find the best buffer arrangements for their vertical wires and save the results together with the solutions. Because we can use a dynamic-programming approach, we can quickly compute the best buffer tree arrangement for a given vertical wire with a number of identical sinks. Consequently we accelerate the search progress for the best buffer tree. The details of implementing the *Vertical Buffering* will be presented in section 4.6.

1.5 Efficient Problem Solving Method

We use a combination of the three recursive structures introduced so far to generate the required subproblems in the feasible region. The correlation of those recursive structures is:

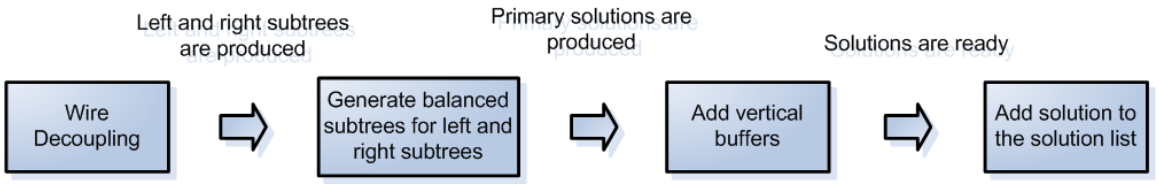


Figure 0-20 How a solution is produced and added to the search space

The procedure of making new subproblems is as follows:

First, two subtrees are generated by wire decoupling: the left and the right subtree. Then for each subtree we take the proper action given its type. For the right subtree, which is always a uniform one, we perform both a new wire decoupling and balanced-buffering[1]. To solve the left decoupled subtree, which is always a non-uniform one, we are only allowed to do balanced-buffering and no more wire decoupling is done. In the end, and before adding the solutions generated by the combination of wire decoupling and balanced-buffering to the search space, the third recursive function, vertical buffering, takes over and performs the buffer insertion on each solution in order to improve its intrinsic delay. In this context the intrinsic delay is defined as the delay observed from the source gate to the solution buffer, taking the interconnect delay into consideration. Vertical buffering is done during solution list creation, while the two other recursive functions are applied in the main algorithm.

Although balanced buffering is permitted for either type of decoupled subtrees, the type of balanced buffering performed on the left subtree differs from the one done for the right subtree. As will be discussed in section 4.7, we apply a specific type of balanced buffering algorithm to solving the left subtree such that the calculated solution does not affect the solution to the right subtree.

1.5.1 Solution Quality

One fundamental assumption in defining any kind of recursive structure applied to making new subproblems is having a group of structurally independent subproblems. One major problem with decoupling a tree is that the left and the right subtrees are not entirely independent, as they share

[1] For more simplicity whenever we talk about balanced buffering, we actually mean the conventional buffer tree construction method introduced in chapters 1 and 2, where we only divide solutions to obtain new subproblems

the same root. In fact the calculated “current delay” for any of those subtrees is affected by the structure of the other one. If for example the input capacitance seen from the first stage of the left subtree increases, the “current delay” at the shared root increases likewise. The larger the “current delay” at the common root, the larger the “current delay” of the right subtree, an undesirable condition that affects the bound values of the right subtree and ultimately its solution quality. To circumvent this drawback, we completely isolate each subtree and solve them separately. To achieve the best solution to each subtree, we temporarily set the “current delay” available at the subtree root to zero. This decision allows the subtree to calculate its best solution regardless of the originating tree structure it belongs to. Computing the best solution to each subtree, the best tree for the originating tree is formed by putting the left and the right best buffer subtrees together. The proposed isolation technique provides a robust basis on which we can implement the wire decoupling method with a guaranteed solution quality. In fact, it will happen that subtrees are faster than necessary, requiring more computing power to achieve. Therefore, we guarantee solution quality at the cost of runtime.

Solving each subproblem independently, two best buffer trees with (possibly) two different best delays are achieved. One important issue is to determine the best delay of the original buffering problem. As the left and the right best buffer trees might present different best delay values, the critical path lies on the subtree with the larger best delay. Therefore the real best delay of the original problem is defined as:

Best Delay = **maximum** (left subtree best delay, right subtree best delay)

To summarize the overall procedure of solving two decoupled subtrees we break the required tasks into 4 steps:

1) Partitioning the tree into the left and right subtrees.
2) Finding the best buffer tree and the best delay of each subtree
3) Combining the left and the right best buffer trees to form the best buffer tree structure of the original problem.
4) Determining the effective best delay of the original problem by detecting the critical path.

1.5.2 Adopting the Wire Decoupling with the Branch-And-Bound Algorithm

In a minimization problem being solved by the branch-and-bound algorithm, one fundamental concept is to have a global variable keeping the minimum upper bound encountered so far[1]. This global variable is used to effectively prune the redundant solutions away from the search space. Since we solve the left and the right decoupled subtrees independently, and regardless of the current delay at their common source gate, the best solutions to the left and right subproblems are obtained locally. Therefore we have to define local best delay variables. This implies that we deal with a number of local buffering problems, while their results must somehow improve the global best delay of the original buffering problem as well.

To resolve this problem, we perform the buffer tree construction hierarchically and at multiple levels. Solving the original buffering problem is done at level zero; the first two buffering subproblems (left and right subtrees decoupled from the original tree) are solved at level 1, and so on. A new buffering level occurs whenever a decoupling solution produces two subtrees. Each buffering level has its own independent best delay and the best solution stack. Once the left and the right subproblems are solved and the real critical path is determined, we update the best delay related to that buffering level, (if, of course, it is required). Propagating up the best solutions to each local buffering problem, the best buffer tree is found in the end for the original buffering problem at level zero. Applying a hierarchical buffering method, the fundamental requirements of implementing a branch-and-bound algorithm can be met while the ideal of wire decoupling is effectively put into practice.

1.6 Vertical Buffering

We improve the delay of each dividing solution before adding it to the search space by performing buffer insertion on the vertical wires. We use a dynamic programming algorithm as the nature of the problem is appropriate. Due to the existence of identical sinks in a dividing solution, we can partition a long wire with multiple sinks into smaller wire sections with a single sink. Because of the symmetrical properties of a balanced tree, we solve the vertical buffering problem only for half of the tree and copy the solution to the other half. We isolate a wire section such that it is connected to a capacitive load at one end, and to a source gate at the other end. Finding the best buffer arrangement, we use the output resistance of the buffering solution to each wire section to do vertical buffering for the next wire section. The improved delay of a dividing solution is the sum of the delays of the buffered wire segments. The final delay is added to the

[1] For more details on the branch and bound method see chapter 1: “The Appropriate Method”.

solution at the time of solution list creation together with the information of the vertically inserted buffers.

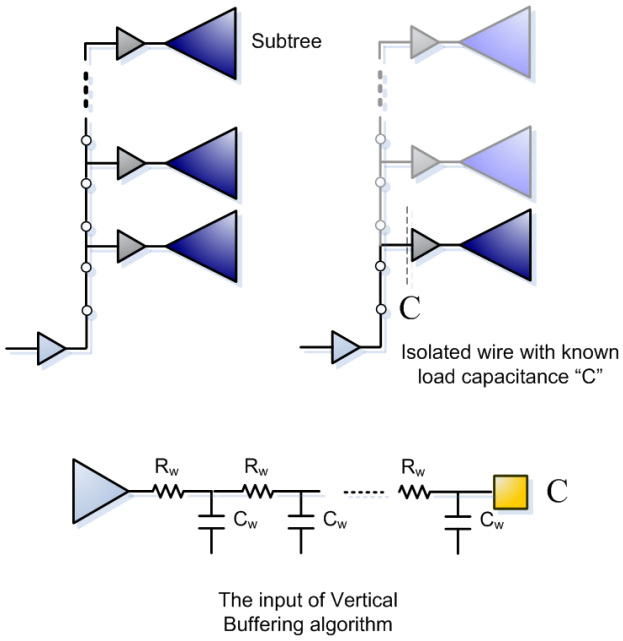


Figure 0-21 Modeling the vertical buffering problem

Temporarily we assume that no legal buffering position can be left empty, which means the final solution has a certain buffer type at every legal buffering position. Having *B* buffers in the buffer library, we transform the vertical buffering problem into a finite state machine (FSM) as follows:

1- the start state is the source gate
2- the acceptance state (also known as end state) is the capacitive load
3- An intermediate state is a buffer from the buffer library
4- Each transition corresponds to adding a new wire segment

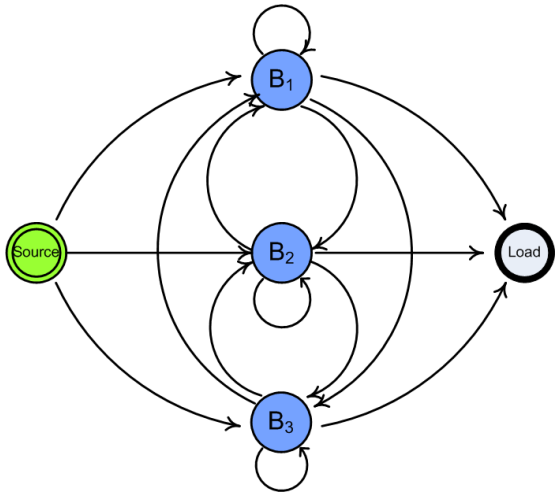


Figure 0-22 FSM corresponding to a vertical buffering problem with 3 buffers

We start with the source gate and add the wire sections as the machine performs state transition. The terminating condition is fulfilled when the machine does L state transitions where L is the number of wire segments. A buffering solution is then a sequence of wire-buffer pairs produced by the machine. One can detect some similarities between the finite state machine shown and a shortest path problem. In both problems we are interested in finding a path with the minimum cost between a starting state and a destination state. The difference is that in this finite state machine we are allowed to meet the same state (the same buffer) multiple times, while in the shortest path problem we have to meet each state only once. In addition, in the finite state machine associated with a vertical buffering problem there is a length constraint to enable a terminating transition, whereas in a typical shortest path problem we do not deal with such restriction. We expand the possible sequences of our finite state machine to reach an optimal solution structure required for solving the problem:

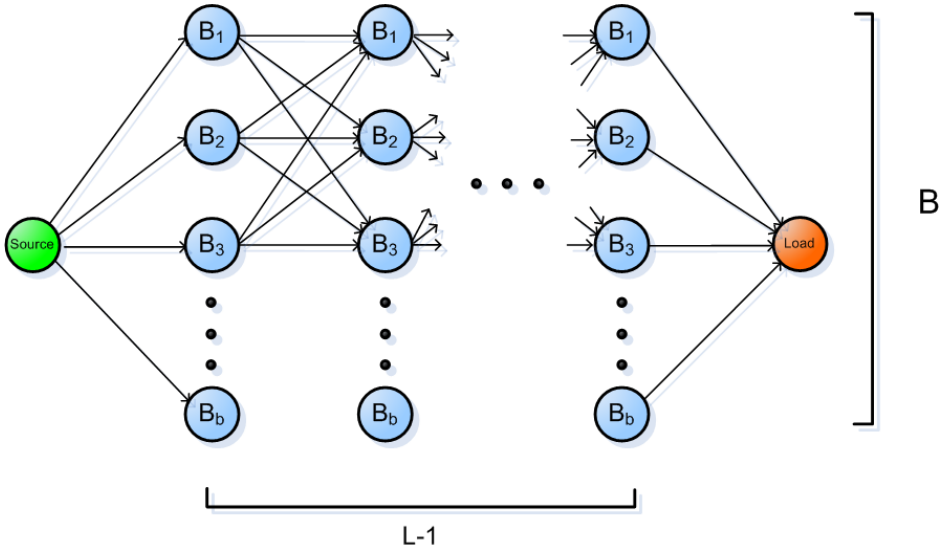


Figure 0-23 The solution table for vertical buffering.
B is the buffer library size and L is the number of wire segments.

This also corresponds to a segmented wire with multiple buffering choices at each buffering position.

Problem Statement: which path must be taken from the source to the load, and through the intermediate buffers, to minimize the delay at the load?

A "brute force" approach to solve the problem would become infeasible as exhaustively working through all possible paths would require $\Omega\left(B^{L}\right)$ time for B buffers and L wire segments. Consequently, we choose a dynamic programming approach to solve the problem.

The first step of realizing the dynamic programming paradigm is to characterize the structure of an optimal solution. Let us consider each node in the figure 4-23 is a legal buffering position that is addressed by node(i,j), for i = 1,2,…,B and j = 1,2,…,L , where B and L are respectively the size of the buffer library and the number of wire sections. We say the fastest buffering sequence from the source to the node (i, j) is achieved through the fastest buffering sequence available at any of the preceding column nodes, (1, j-1), (2, j-1),..., (B, j-1), plus the wire connecting the node (i, j) to its best preceding node.

In order to determine the value of an optimal solution recursively, we define a recursive cost function. Let $c_i[L]$ be the fastest buffering sequence from the source through the buffer type i at the last buffering position L, and let d_i be the interconnect delay resulting from the wire connecting the buffer type i at the last buffering position to the load. The cost function of the last node f^* is defined as:

$$f^* = \min\left(c_1[L]+d_1, c_2[L]+d_2, ..., c_B[L]+d_B\right)$$

To have the cost function of the nodes at the first level, we define e_i as the wire delay between the source and the i_{th} nodes at the first column, and α_i as the internal cost of each node, which corresponds to the intrinsic delay of the i_{th} buffer.

$$f_1[1] = e_1 + \alpha_1$$

$$f_2[1] = e_2 + \alpha_2$$

$$\vdots$$

$$f_B[1] = e_B + \alpha_B$$

Now to compute the cost function for j = 2, 3, …, L , let $f_i[j]$ denote the fastest buffering sequence from the source through node (i, j), we have:

$$f_i[j] = \min\left(f_1[j-1]+w_{1,i}, f_2[j-1]+w_{2,i}, ..., f_B[j-1]+w_{B,i}\right)+\alpha_i$$

Where:

i is the row index of the current node and also corresponds to the i_{th} buffer of the buffer library

j is the column index of the current node and also corresponds to the j_{th} wire segment

B is the size of buffer library

α_i is the internal cost of the current node and also corresponds to the intrinsic delay of the current buffer

$w_{a,b}$ is the cost value between node a from the previous column and node b of the current column. It also corresponds to the interconnect delay between two buffers a and b with a unit wire segment in between.

The Algorithm

Observe that in the cost function each value of $f_i[j]$ depends only on the values of one preceding column. By computing the node values in a left to right order, the best delay is achieved at the last node. Having computed all node values, the best buffering arrangement is achieved by constructing the best path from the load to the source.

Fastest Buffering Sequence (S*ource Gate, Buffer Library, Load Capacitance, Wire Segments*)

1 **for** i = 1 to #Buffers in Library
2 **do** $f_i[1] = e_i + \alpha_i$
3 **for** j = 2 to L // *number of wire segments*
4 **for** i = 1 to #Buffers in Library
5 **do** $f_i[j] = \min\left(f_1[j-1]+w_{1,i}, f_2[j-1]+w_{2,i}, ..., f_B[j-1]+w_{B,i}\right)+\alpha_i$
6 node (i, j) ->Save Best Parent
7 $f^* = \min\left(c_1[L]+d_1, c_2[L]+d_2, ..., c_B[L]+d_B\right)$

//Making the solution path

8 temp parent = last node->best parent
9 Solution Stack->Push (temp parent)
10 **for** j = L to 1
11 **do** temp parent = temp parent->best parent
12 Solution Stack->Push (temp parent)

Figure 0-24 Vertical Buffering algorithm

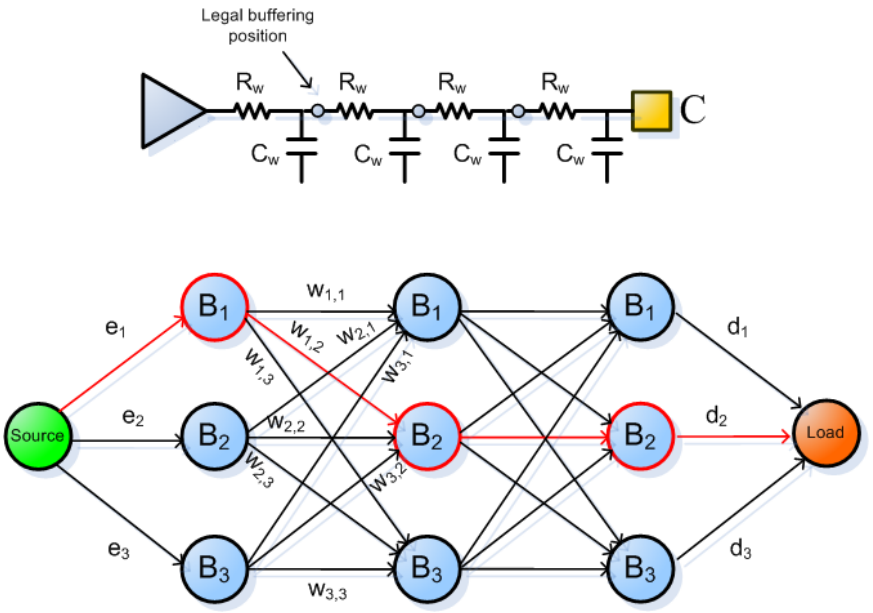
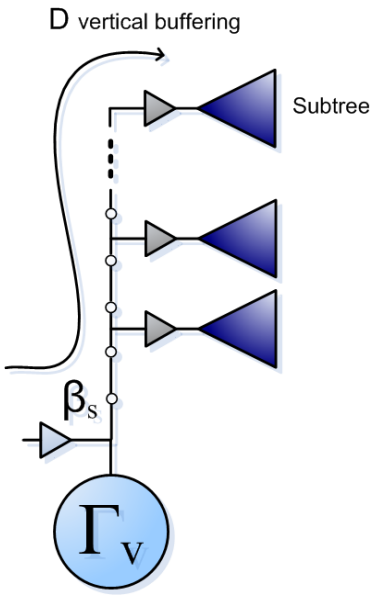


Figure 0-25 Example of solving the Vertical Buffering problem.
It is solved for a 4-segment wire, using a buffer library of size 3.

In order to find the best buffering solution to a long wire with multiple fanouts we divide the wires into smaller single fanout wires and call procedure *Fastest Buffering Sequence* to compute the best buffer arrangement. Each buffering solution to a wire segment presents a new source gate to the next wire segment. The total delay of the buffered vertical wire is achieved as:



$$Delay_{total} = \left(\sum_{n:NumberOfWireSegments} BestDelay_n \right) + (\beta_S . \Gamma_V)$$

Figure 0-26 Calculating the total delay of the buffering solution

Where β_S is the output resistance of the original source gate and δ_V is the input capacitance seen from the vertical buffers at the source gate. Note that *n* is the number of wire sections in the upper half of the tree for which we do vertical buffering.

Empty Buffering Choices

Since the best buffering solution to a vertical wire may contain some relatively long non-buffered wires, forcing a legal buffering position to be always filled with buffer could lead to sub-optimal solutions. On the other hand, we would like not to complicate the optimal solution structure that has been defined previously. As a compromise, we introduce some virtual buffers representing empty buffering choices, called *Pseudo Buffers*. For each buffer in the library there exists a pseudo buffer such that inserting the pseudo buffer on the wire segment driven by the real buffer does not change the total interconnect delay value. To have a pseudo buffer, we define its input capacitance, output resistance and intrinsic delay as follows:

γ_{Pseudo} (Input capacitance) = 0

α_{Pseudo} (Intrinsic delay) = 0

β_{Pseudo} (Output resistance) $= \beta_{OriginatingBuffer} + r$, where $\beta_{OriginatingBufer}$ is the output resistance of its derived buffer and *r* is the wire unit resistance

The delay equivalence of two different configurations, one with pseudo and one without pseudo buffer, is shown in the following figure.

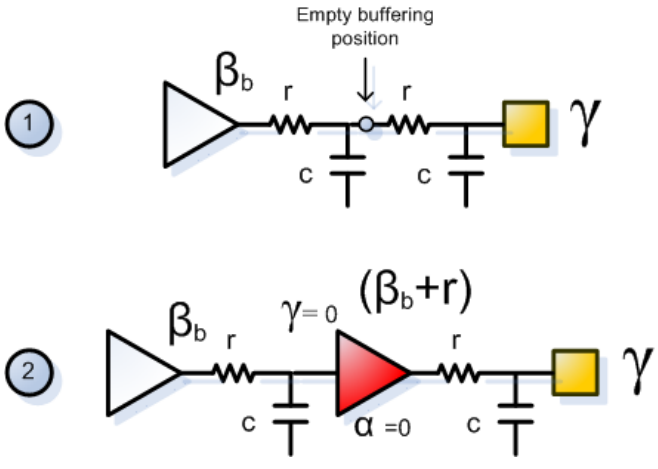


Figure 0-27 Inserting a pseudo buffer at an empty legal position

$$D_1 = (\beta_b + r)(2c + \gamma) + r(c + \gamma)$$

$$D_2 = (\beta_b + r)c + (\beta_b + r)(c + \gamma) + r(c + \gamma)$$

$$\Rightarrow D_1 = D_2$$

Formula Notation:

D : Delay β_b :Buffer output resistance r :Wire unit resistance

c :Wire unit capacitance γ :Load capacitance

There are some critical considerations in using a pseudo buffer in our Dynamic Programming problem solving method:

1- Each pseudo buffer can only be connected either to itself or its originating buffer at the previous level.

2- If the minimum value of the cost function suggests that the current pseudo buffer be connected to itself at the preceding column, its output resistance must be modified to consider physical properties of the additional piece of wire. Following the definition of a pseudo buffer, its output resistance is always the sum of the output resistance of its originating buffer and the wire unit resistance. The intrinsic delay and the input capacitance of the pseudo buffer remains unchanged as they are set to zero. As a result, the output resistance of a pseudo buffer after n_{th} wire segment from its originating buffer would be:

$$\beta_{n_{th}PseudoBuffer} = \beta_{OriginatingBufer} + n.r_{wire}$$

3- We also make a pseudo buffer specifically for the original source gate in order for our algorithm to cover all possible solutions. Such a pseudo buffer can only be preceded by itself.

4- Obviously only the pseudo buffer of the source gate is allowed to be considered at the first buffering stage, since there is no buffer at the last stage.

5- Those pseudo buffers which are associated with available buffers in the buffer library are allowed to be considered from the 2nd buffering stage.

6- Every normal buffer can be preceded by any pseudo buffer.

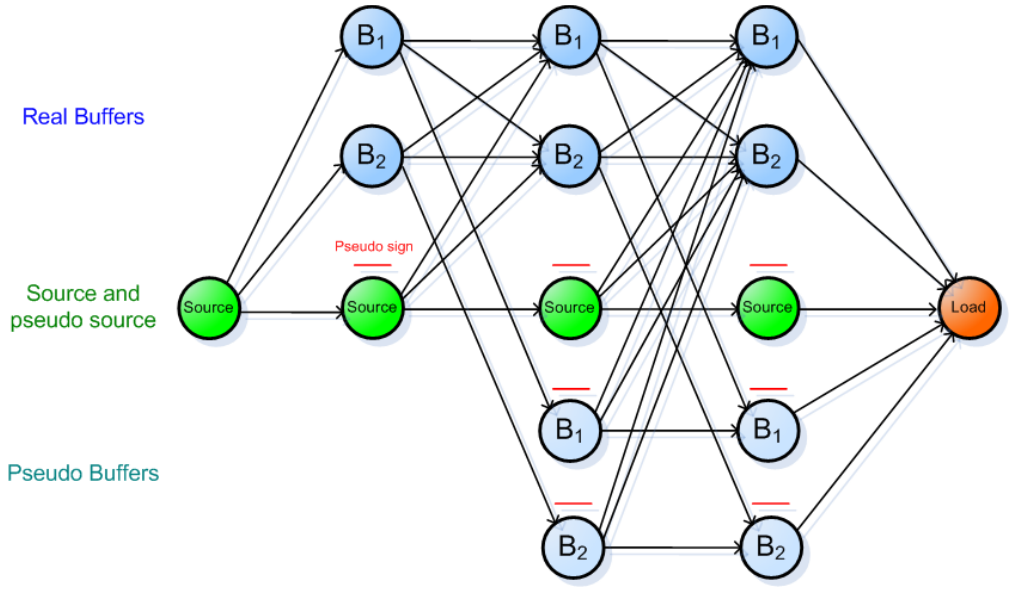


Figure 0-28 Example of a solution table with pseudo buffers.
It is constructed for a 4-segment wire, using a buffer library size of 2.

We now modify the algorithm to also include the empty buffering choices through pseudo buffers:

Fastest Buffering Sequence (*Source Gate, Buffer Library, Load Capacitance, Wire Segments*)

1 make Pseudo Buffer of the source gate and add it to the library
2 **for** i = 1 to #Buffers in Library
3 **do** $f_i[1] = e_i + \alpha_i$
4 **for** i = 1 to #Buffers in Library
5 **do** make Pseudo Buffer of the i_{th} buffer and add it to the library
6 **for** j = 2 to L // *number of wire segments*
7 **for** i = 1 to #Buffers in Library
8 **if** (the current node is a normal buffer)
9 $f_i[j] = \min\left(f_1[j-1]+w_{1,i}, f_2[j-1]+w_{2,i}, \ldots, f_B[j-1]+w_{B,i}\right)+\alpha_i$
10 node (i, j) ->Save Best Parent
11 **else if** (it is a normal pseudo buffer)
12 $f_i[j] = \min\left(f_i[j-1]+w_{i,i}, f_{OriginatingBuffer}[j-1]+w_{OriginatingBuffer,i}\right)$
13 node (i, j) ->Save Best Parent
14 **if** node (i, j) -> Best Parent = = itself
15 **do** modify its output resistance
16 **else** // it is the source gate pseudo buffer
17 $f_i[j] = f_i[j-1]+w_{i,i}$
18 node (i, j) ->Save Best Parent

19 modify its output resistance

20 $f^* = \min\left(c_1[L]+d_1, c_2[L]+d_2, ..., c_B[L]+d_B\right)$

21 temp parent = last node->best parent

22 Solution Stack->Push (temp parent)

23 **for** j = L to 1
24 **do** temp parent = temp parent->best parent

25 Solution Stack->Push (temp parent)

Figure 0-29 Vertical Buffering algorithm using pseudo buffers

Handling Phase Shifting in the Presence of Inverting Buffers

Due to the fact that our buffer library may consist of some inverting buffers, the vertical buffering function can produce some negative phase solutions. As the objective of vertical buffering is only to improve the delay of dividing solutions, we prefer to avoid vertical buffering arrangements that result in any phase shifting. The vertical buffering function is consequently done with the constraint of only producing positive phase solutions. To that effect, we must build up the buffered wire with positive phase sub-solutions.

Considering the inverting buffers, there always exist two fastest paths from the source gate through an arbitrary buffering position: one with negative phase, and one with positive phase. As the table of solutions is filled from the source to the load, we calculate and save both the fastest paths and pass them to the next column. Accordingly, at each node in the solution table, two minimum costs with two best parent nodes must be maintained. Selecting the fastest positive phase solution by the load at the last column, the ultimate positive phase solution is constructed, while it may contain some inverting buffers within it. Moreover, we have to create an additional pseudo buffer for each buffer in the library to enable it to transmit its best negative phase solution as well the positive phase one. Yet the type of pseudo buffer is still positive-phase since it is representing a single wire with no phase shifting.

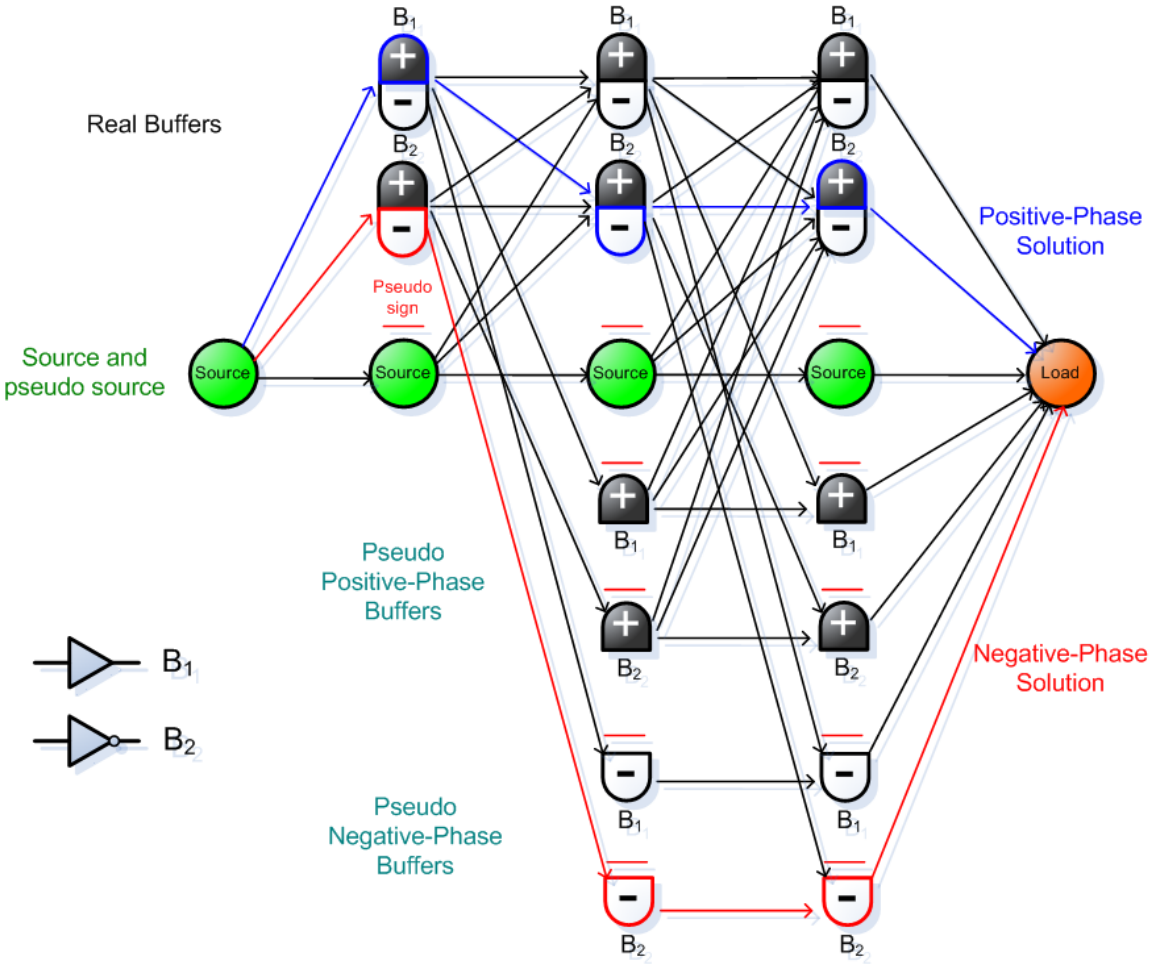


Figure 0-30 Example of producing negative-phase and positive-phase solutions.

Fastest Buffering Sequence (*Source Gate, Buffer Library, Load Capacitance, Wire Segments*)

1 make Pseudo Buffer of the source gate and add it to the library

2 **for** i = 1 to #Buffers in Library

3 **do if** (current node has inverting buffer)

4 $f_i[1]_+ = \infty$

5 $f_i[1]_- = e_i + \alpha_i$

6 **else**

7 $f_i[1]_+ = e_i + \alpha_i$

8 $f_i[1]_- = \infty$

9 **for** i = 1 to #Buffers in Library

10 **do** make negative and positive phase Pseudo Buffers of the i_{th} buffer and add it to the library

11 **for** j = 2 to L // *number of wire segments*

12 **for** i = 1 to #Buffers in Library

13 **if** (the current node is a normal buffer)

14 **if** (current node has inverting buffer)

15 $$f_i[j]_+ = \min\left(f_1[j-1]_- + w_{1,i}, f_2[j-1]_- + w_{2,i}, ..., f_B[j-1]_- + w_{B,i}\right) + \alpha_i$$

16 node (i, j) ->Save Best Positive Parent

17 $$f_i[j]_- = \min\left(f_1[j-1]_+ + w_{1,i}, f_2[j-1]_+ + w_{2,i}, ..., f_B[j-1]_+ + w_{B,i}\right) + \alpha_i$$

18 node (i, j) ->Save Best Negative Parent

19 **else** // current node has non-inverting buffer

20 $$f_i[j]_+ = \min\left(f_1[j-1]_+ + w_{1,i}, f_2[j-1]_+ + w_{2,i}, ..., f_B[j-1]_+ + w_{B,i}\right) + \alpha_i$$

21 node (i, j) ->Save Best Positive Parent

22 $$f_i[j]_- = \min\left(f_1[j-1]_- + w_{1,i}, f_2[j-1]_- + w_{2,i}, ..., f_B[j-1]_- + w_{B,i}\right) + \alpha_i$$

23 node (i, j) ->Save Best Negative Parent

24 **else if** (it is a normal pseudo buffer)

25 **if** (it is made for positive phase path)

26 $$f_i[j]_+ = \min\left(f_i[j-1]_+ + w_{i,i}, f_{OriginatingBuffer}[j-1]_+ + w_{OriginatingBuffer,i}\right)$$

27 $$f_i[j]_- = \infty$$

28 node (i, j) ->Save Best Positive Parent

29 **if** node (i, j) -> Best Parent = = itself

30 **do** modify its output resistance

30 **else** //it is made for negative phase path

32 $$f_i[j]_- = \min\left(f_i[j-1]_- + w_{i,i}, f_{OriginatingBuffer}[j-1]_- + w_{OriginatingBuffer,i}\right)$$

33 $$f_i[j]_+ = \infty$$

34 node (i, j) ->Save Best Negative Parent

35 **if** node (i, j) -> Best Parent = = itself

36 **do** modify its output resistance

37 **else** // it is the source gate pseudo buffer

38 $$f_i[j]_+ = f_i[j-1]_+ + w_{i,i}$$

39 $$f_i[j]_- = \infty$$

40 node (i, j) ->Save Best Parent

41 modify its output resistance

42 $$f^* = \min\left(c_1[L]_+ + d_1, c_2[L]_+ + d_2, ..., c_B[L]_+ + d_B\right)$$

43 temp parent = last node->best parent

44 Solution Stack->Push (temp parent)

45 Current node = temp parent

46 **for** j = L to 1

47 **do**

48 **if** (current node has non-inverting buffer)

```
49                  if (the best positive phase path was used by the previous stage)
50                          temp parent = current->best positive parent
51                  else
52                          temp parent = current->best negative parent
53          else // current node has Inverting buffer
54                  if (the best positive phase path was used by the previous stage)
55                          temp parent = current->best negative parent
56                  else
57                          temp parent = current->best positive parent
58          current node = temp parent
59          Solution Stack->Push (temp parent)
```

Figure 0-31 Vertical Buffering algorithm considering inverting buffers

Vertical Buffering Time Complexity

We now study the time complexity of the algorithm designed. As the program runtime is spent mainly in computing the cost functions, we study how much time it generally takes for a node to obtain its cost values. Every node must read the positive-phase and the negative-phase best delays values of all real buffers from the previous stage, together with the cost function of all pseudo buffers. Given a buffer library with *B* buffers, every node reads:

1- 2**B* nodes from the previous column. Each node is read twice as it has two minimum costs, one for the positive phase fastest path, and one for the negative phase fastest path.
2- *B* pseudo nodes carrying a positive phase path
3- *B* pseudo nodes carrying a negative phase path
4- One pseudo node directly connected to the source gate

The sum of all access nodes is $\sum NodeAccess = 4B + 1$

As at each column there are *B* nodes for *B* real buffers, we obtain the total node access for all of them:

$$\sum_{\text{RealBuffers}} (4B + 1) = B(4B + 1)$$

Each pseudo buffer also reads two nodes from the previous stage, itself and its originating buffer, except for the pseudo buffer made for the source gate, which reads only itself.

$$\sum_{\text{Real Buffers + Pseudo Buffers}} B\left(4B+1\right)+2B*2+1=4B^2+5B+1$$

This makes the algorithm run in $O\left(B^2\right)$ time for each column. Having L columns in the cost table corresponding to L wire segments, the complete time complexity is $O\left(LB^2\right)$, which is generally satisfying as the number of buffers in the library does not exceed a few dozens, and the impact of the wire length appears to be linear on the vertical buffering runtime.

1.7 Memory Reuse

Thus far, we have studied three different structures that recursively generate the subproblems which form the search space. Now we would like to know if there exists any common sub-subproblem in the search space created by these recursive functions. We will then be able to effectively avoid redundant calculations and prune the non-promising solutions from the solution list. Looking deeper into each recursive structure, the existence of such common sub-solutions is clearly recognized:

A) Wire Decoupling: Repeatedly producing and solving local buffering subproblems, wire decoupling generates a considerable amount of common sub-solutions. The existence of common sub-solutions comes from the nature of the decoupling operation where each uniform subtree is also allowed to be decoupled. In the following example a decoupled right subtree appears twice in the decoupling process:

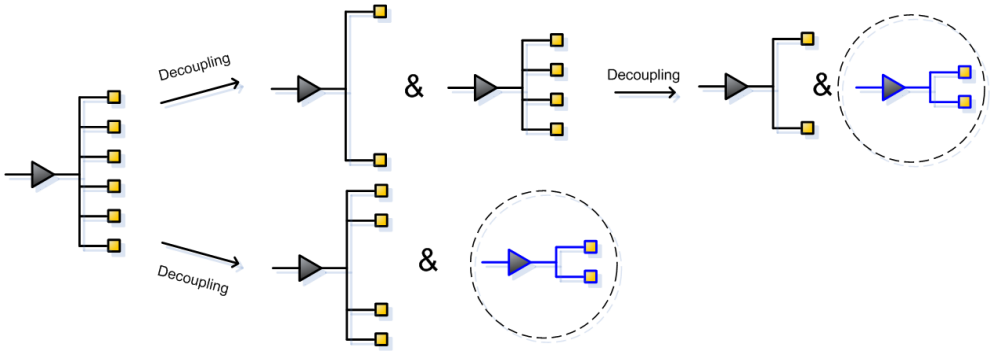


Figure 0-32 Similar decoupled subtrees

If we save and retrieve the best buffer tree of a decoupled subtree, further references to the previously calculated solution become possible.

B) Traditional Balanced Buffering: By *Traditional Balanced Buffering* we mean the buffering method introduced in the first and the second chapter. The issue of common sub-solutions was extensively addressed before and the underlying concepts and data structures provided in the last two chapters can be reutilized to effectively improve the program runtime.

C) Vertical Buffering: Due to the nature of *Vertical Buffering*, we expect to have a large number of subproblem recalculations in the search space. However, we do not complicate our algorithm by implementing memory-reuse methods for vertical buffering because its time complexity is relatively small.

The following sections explain the necessary modifications needed for the new algorithm.

1.7.1 Solutions

We deal with three solution types, each of which associated with one recursive subproblem generator:

a) **Dividing Solutions**: using a branching factor, divide a tree (whether uniform or non-uniform) into balanced subtrees.

b) **Decoupling Solutions**: based on a boundary value, they suggest how a tree can be decoupled into two subtrees. Decoupling solutions are only allowed to be applied to uniform trees.

c) **Vertical Wire Buffering Solutions**: The vertical buffering function computes a buffering solution to the vertical wire of the recently enumerated dividing solution. Therefore, this type of solutions is constructed during solution list enumeration and is saved inside the dividing solution class.

1.7.2 Solution Lists

A solution list normally contains only two solution types, dividing and decoupling, as *vertical wire buffering* solutions are not visible to the solution list. Yet, not all solution lists can contain decoupling solutions. Given the fact that non-uniform trees are not allowed to be decoupled, the subproblems associating with non-uniform trees are allowed to only make dividing solutions. Thus before any solution list enumeration, we should identify the type of the tree for which we are going to make solutions. We classify the solution lists also into two groups:

1- Complete Solution Lists
2- Limited Solution Lists

A complete solution list may contain both solution types, and is always enumerated for a subproblem with a uniform tree. In contrast a limited solution list is constructed for a subproblem with a non-uniform subtree and only consists of dividing solutions. A limited solution list has two main differences with a complete one: first, it does not include any decoupling solution; second, the divisor value of each dividing solution must be chosen such that it does not affect the best solution to the right subtree. For example a branching factor of 3 is not acceptable for a left subtree with a fanout number of 8, since it produces some subtrees that share the same source buffer with the right subtree. In addition, such a divisor does not lead to any balanced subtree. This is illustrated in Figure 4-33.

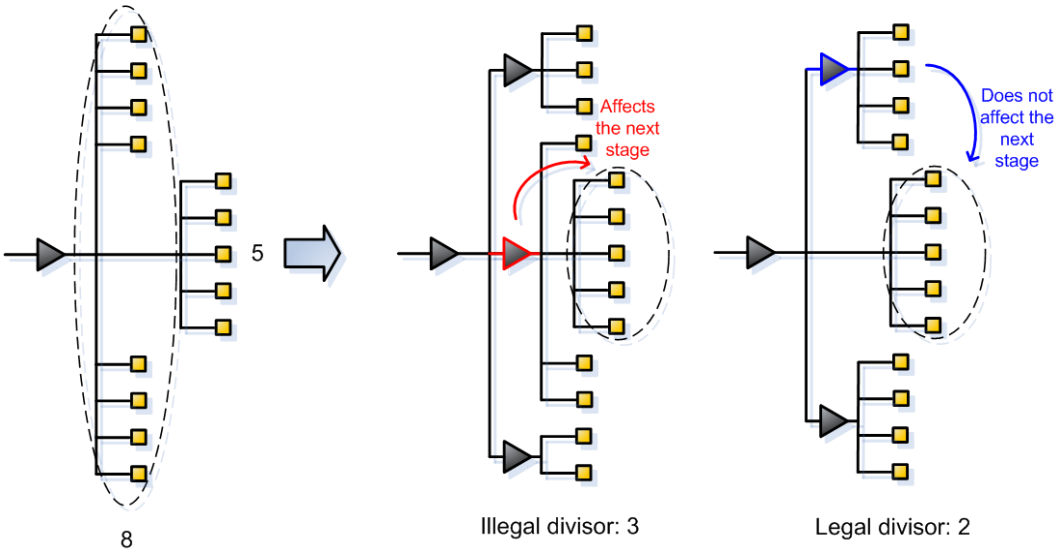


Figure 0-33 Legal and illegal divisors of a left subtree with a fanout of 8

To have the safest set of divisors suitable for making the dividing solutions of a limited solution list, we always generate and save the divisors for half of the left subtree. In addition, we forbid any branching factor of 1, as it places a buffer at the junction of two subtrees.

1.7.3 Real Delay Value Field

In order to make it possible to save and recover solution results, we also add a data field to the decoupling solutions to store any real delay value. That data field is set whether at the time of solution list creation or when the search process is in the main algorithm. We deal with the issue of updating the real delay of a decoupling solution as for the dividing solutions in chapter two. The details of the required implementations will be shown in section 4.9.

1.7.4 Solution Ranking

In making a complete solution list, where a range of decoupling and dividing solutions is available, one critical issue is the solution list traversal order. Putting the decoupling solutions at the top of the list would orient the search direction toward late updates in the best delay. Given the fact that decoupling solutions merely helps getting more possibilities in the search space and never results in a buffering solution, we put the dividing solutions first in the solution list in order for our search progress to converge in a reasonable amount of time.

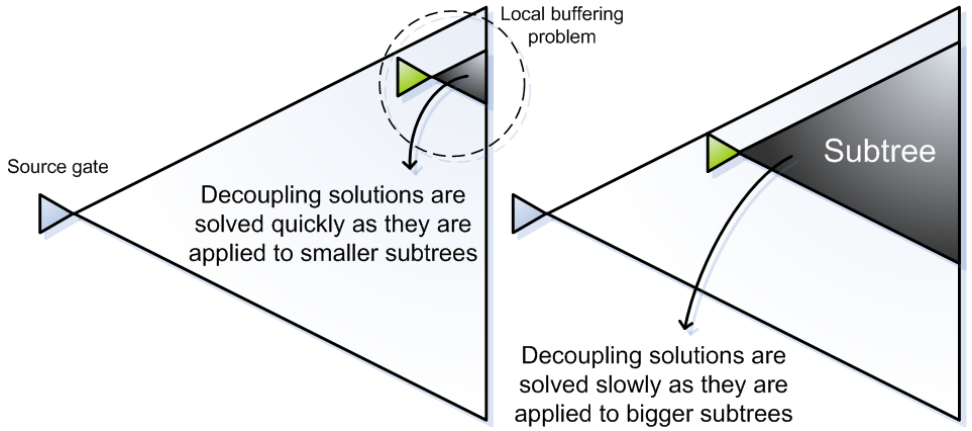


Figure 0-34 Solution convergence

1.7.5 Solution Stack inside the Solution List

To facilitate any kind of subproblem solution retrieval at the time of saving the best buffer tree, we add a solution stack to the solution list to store the best subproblem solution. We save a copy of the best solution found by the main algorithm as soon as it becomes available, and safely use it whenever needed. The necessary and sufficient condition of doing so is to have a solution list with all solutions entirely solved. Filling up the solution stack is done either at the time of solution list creation or during the search procedure in the main algorithm.

1.7.6 Pruning

We apply the same list relaxing technique to reduce the size of our solution lists as the method we introduced in the second chapter. The only difference between the old pruning function and the new one is the way decoupling solutions are handled. As we sort a complete solution list such that the decoupling solutions are placed at the bottom of the list (below the dividing solutions), the pruning function does not eliminate those solutions. Therefore, one has to modify the pruning function to ignore the unsolved decoupling solutions placed at the bottom of the list.

1.7.7 Solution Lists Access Method

As we intend to save and reuse the previously calculated solutions, we need to devise an efficient data structure in order to maintain the solution list and access them in a reasonable amount of time. We employ the Lazy Weight Binary Search Tree introduced in chapter 2 with one modification. The access key used in implementing the binary search tree in chapter 3 previously consisted of 2 fields: fanout number and buffer type and with a 2-dimensional coordinate system the search space was successfully covered. Now we need to add one more dimension to the access key: Δ.

We introduced earlier the fanout encoding technique and we argued that two subproblems with the same fanout number can be differentiated based on their axis of symmetry and Δ. As the structure of the best buffer tree is not sensitive to the axis of symmetry, two subproblems with identical Δ and identical fanout numbers can share the best solution even if they have different axis of symmetries. As a result, we ignore the axis of symmetry in developing the solution list access method. What remains to produce the access key is fanout number, Δ and buffer type. Now we have a 3-dimensional coordinate system to cover all possible subproblems.

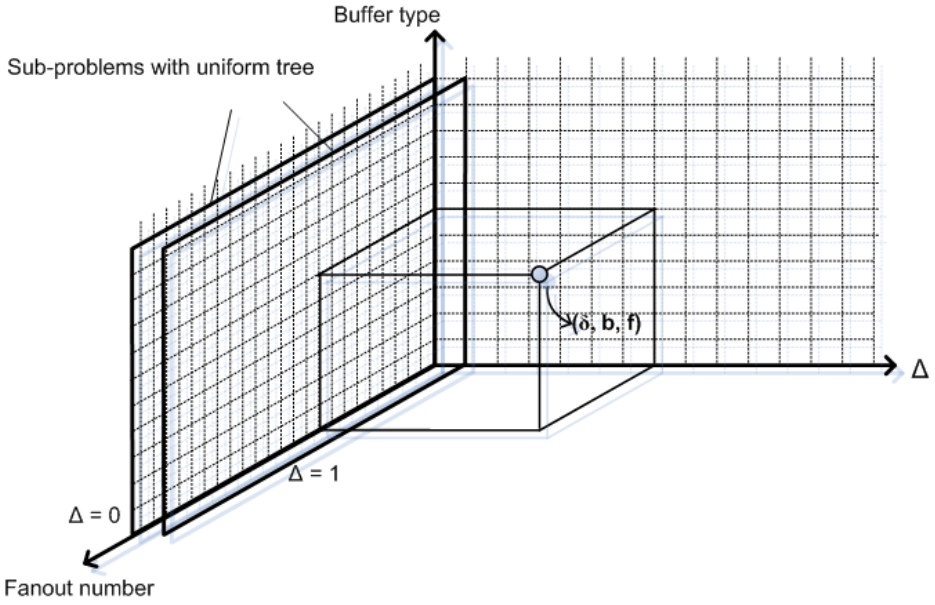


Figure 0-35 3-dimensional coordinate system used to maintain the solution lists

Recursively generated by the wire decoupling method, a large number of subproblems with Δ>1 can be expected for a typical buffering problem, large enough to use a separate Lazy Weight Binary Search Tree to maintain their solution list addresses. In order to implement a 3-dimensional data structure for storing the solution list, we use the binary search tree presented in chapter 2 and replace the solution list address field of the look-up table with the root address of a

Lazy Weight Binary Search Tree. The encapsulated search tree has a key access of Δ and maintains only one solution list address inside each node.

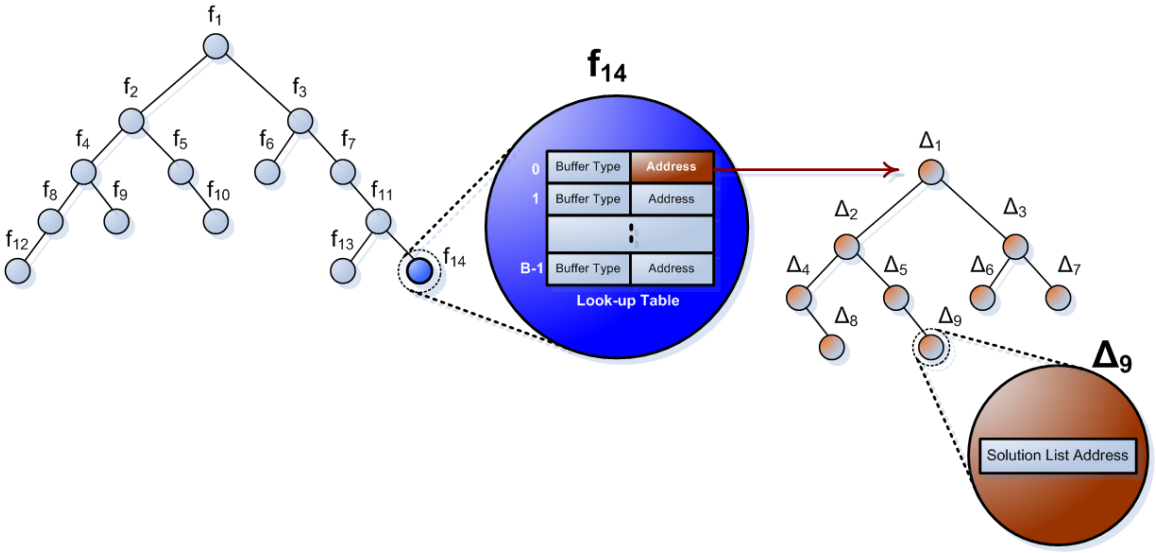


Figure 0-36 A 3-level data access method

The search tree manipulations techniques, including self-reorganizing operations, are done similarly to what we explained in chapter 2.

1.8 Modified Smart Bound

In the chapter 3 we introduced a method to boost the bound and called it smart bound. In order to adopt the same technique for the balanced buffering method presented in this chapter we perform a slight modification on the old smart bound.

In the original method, we saved the best "current delay" value in the solution list and updated it whenever the same subproblem received a better "current delay". The bound equation and solution filtering condition

$$Bound = D_{current} + \alpha_b + SubTree_{IdealDelay}$$

If (Bound > Best Delay): Do not solve the subproblem

A solution list might be accessed through different local buffering problems with different local best delays; we would like to know how we can decide whether to solve or ignore a subproblem which is frequently met by the search traversal with different local "current delay"s and different local "best delay"s. A simple approach is to save the "best delay" in the solution list and monitor

the course of “current delay” and “best delay” changes together; then under the following circumstances we let the subproblem be solved. If we have:

$$\delta D_{current} = D_{current_{\text{RECENT}}} - D_{current_{\text{OLD}}}$$

$$\delta BestDelay = BestDelay_{\text{RECENT}} - BestDelay_{\text{OLD}}$$

we say solving a subproblem does not lead to any better solution if:

$$\delta BestDelay < \delta D_{current}$$

This means that if the filtering condition has previously refused to solve a subproblem, it would definitely refuse to solve the subproblem for the second time, as the inequality still holds its previous state. In other words, if the “best delay” increase is less than the “current delay” increase, the bound condition still holds:

1.9 Main Algorithm

Our algorithm builds and explores the search space of all possible buffer trees, except those which contain buffered horizontal wires. Before the search traversal begins to explore the search space, a global buffering stack is created to maintain the information associated with each local buffering problem being solved by the algorithm. Whenever a decoupling solution is observed, two new local buffering problems are constructed and pushed into the global buffering stack. Each stack element consequently represents a certain level at which we search for the best local buffer tree. Acquiring two best buffer trees for each decoupling solution, we remove the results of two solved local buffering problems from the stack and use their results to update the best solution of the local buffering problem one level lower. Through the search space traversal the global buffering stack grows larger whenever a decoupling solution is met and grows smaller whenever a decoupling solution is completely solved. Observe that dividing solutions result in no size change in the global buffering stack.

Solution Balanced-Buffering (*source gate, fanout, current delay, current phase*)

```
1   D = current delay + source-gate.intrinsic-delay+ VerticalBuffering (source-gate, fanout)
2    if (fanout.Δ <=1)
3       if ((D < Buffering-stack.top-element.best-delay) and (current phase== positive))
4            then Buffering-stack.top-element.best-stack= Buffering-stack.top-element.solution-stack
5                 Buffering-stack.top-element.best-delay = D;
6       bound = current delay + source-gate.intrinsic-delay + Ideal Delay (source-gate, fanout )
7   if ((fanout.Δ <=1 and bound < Buffering-stack.top-element.best-delay) or fanout.Δ >1)
8       then  make solution list (source-gate, fanout)
9       while (there are solutions in the solution list)
10          Solution = get a solution
11          if (Solution == dividing)
12              then if (Solution does not have real delay)
13                       new fanout = fanout / Solution.branching-factor
14                       new source = Solution.proposed-buffer
15                       new current delay = current delay + Solution.intrinsic-delay
16                       new phase = current phase* Solution.proposed-buffer.phase
17                       Buffering-stack.top-element.solution-stack.push(Solution)
18                       Best-solution = Balanced-Buffering (new source, new fanout, new current
                                                                          delay, new phase)
19                       Buffering-stack.top-element.solution-stack.pop()
20                       if (Best-solution has real delay)
21                           then Solution.real-delay= Best Solution.real-delay
22          else // Solution == Decoupling
23              Local-best-delay= Buffering-stack.top-element.best-delay
24              Buffering-stack.top-element.solution-stack.push(Solution)
25              Left-subtree = DecoupleLeft(fanout,Solution)
26              Left-buffering-problem = MakeLocalBufferingProblem(Solution, Left-subtree)
27              Left-buffering-problem.best-stack.push(VerticalBuffering (source-gate, Left-subtree))
28              Buffering-stack.push(Left-buffering-problem)
29              Best-solution = Balanced-Buffering (current source, Left-subtree, 0, current phase)
30              Right-subtree = DecoupleRight(fanout,Solution)
31              Right-buffering-problem = MakeLocalBufferingProblem(Solution, Right-subtree)
32              Right-buffering-problem.best-stack.push(VerticalBuffering (source-gate, Right-subtree))
33              Buffering-stack.push(Right Buffering Problem)
34              Best-solution = Balanced-Buffering (current source, Right-subtree, 0, current phase)
35              Buffering-stack.pop();
36              Buffering-stack.pop();
37              Left-input-capacitance = Left-buffering-problem.best-solution.input-capacitance
```

```
38              Right-input-capacitance= Right-buffering-problem.best-solution.input-capacitance
39              DelayPath-1=current delay+source-gate.output-resistance* Left-input-capacitance
                                                    + Right-buffering-problem.best-delay
40              DelayPath-2=current delay+source-gate.OutputResistance* Right-Input-Capacitance
                                                    + Left-buffering-problem.best-delay
41              Critical-path = MAX(DelayPath-1, DelayPath-2)
42              if (Critical-path < Local-best-delay )
43                then Buffering-stack.top-element.solution-stack.push(Left-buffering-problem.Best-tree)
44                     Buffering-stack.top-element.solution-stack.push (Right-buffering-problem.Best-tree)
45                     Buffering-stack.top-element.best-stack= Buffering-stack.top-element.solution-stack
46                     Buffering-stack.top-element.solution-stack.pop (Left-buffering-problem.Best-tree)
47                     Buffering-stack.top-element.solution-stack.pop (Right-buffering-problem.Best-tree)
48              Buffering-stack.top-element.solution-stack.pop ()
49      Resort the solution-list
50      Prune the solution-list
51      if (solution-list.first-solution has real delay)
52          then SetSolutionListStack
53      if (there is any real solution in the solutions list)
54          then Real-solution = best real solution of the solution list
55                current delay + source-gate.intrinsic-delay + Real-solution.real-delay
56                If (Buffering-stack.top-element.best-delay)
57                    then Buffering-stack.top-element.solution-stack.push(Solution-list.best-result)
58                         Buffering-stack.top-element.best-stack=Buffering-stack.top-element.solution-stack
59                         Buffering-stack.top-element.solution-stack.push(SolutionListStack)
60                         Buffering-stack.top-element.best-delay = D
61      Return the first solution of the solution list
62  else
63      Return empty solution; //no solution list has been built or explored due to the bound failure
```

Figure 0-37 Balanced Buffering algorithm considering interconnect delay

Algorithm Analysis

Given a source gate, a class containing the fanout information, the delay up to the root of the current subproblem, and the current buffer tree phase, the Balanced-Buffering algorithm launches the search for the best buffer tree. On line 1 the subproblem root is connected to sinks in order to calculate the total delay of current buffer tree. *Vertical Buffering* function is also called to partially reduce the computed delay by adding necessary buffers to the last stage of the current buffer tree. On lines 2 through 5 we perform a best solution update for only uniform trees,

followed by a bound calculation. On line 7 we solve the buffering problem if the current tree is a non-uniform tree, or the current tree is a uniform one but is not filtered by the bound. On line 8 the solution list is created. The search procedure occurs on lines 9 through 48 in a while-loop. At each round, a solution is selected from the solution list. If it is a dividing solution, new fanout and new phase are calculated, and together with the updated current delay and the new source gate are set as the arguments of Balanced Buffering function. During the time it is being solved, the selected solution is kept inside a temporary solution stack belonging to the current local buffering problem, and is removed from the stack whenever the search traversal terminates. On lines 20 and 21 the solution receives a real delay value if it has been completely solved. A decoupling solution is solved on lines 22 through 48. We temporarily save the best delay of the current local buffering problem on line 23, which makes it easier to access that value in further references. On line 24 the solution is pushed in a temporary stack belonging to the current buffering problem, and is removed from the stack on line 48. Lines 25 through 29 create and solve a local buffering problem for the left decoupled subtree. The same solving procedure repeats for the right decoupled subtree on lines 30 through 34. Note that the "current delay" value is set to zero in solving both local buffering problems since we solve them independently. On lines 35 and 36 the global buffering stack is cleaned out from two local buffering problems after they are solved. The input capacitance of the best buffer trees obtained for the left and the right decoupled subtrees are calculated and used in delay path calculations. On line 41 the critical path is determined and if necessary an update to the best solution of the current local buffering problem occurs afterwards. On lines 49 and 51 the solution list is resorted with respect to likely new delay values and, if applicable, is pruned from any redundant solution. If the current subproblem has been completely solved, its solution stack is filled up on line 52. Lines 53 through 60 examine the solution list for any unexplored solution which could have received its real delay at the time of solution list creation and not by performing Balanced-Buffering. If there is any, the solution with the minimum real delay in the solution list is detected and a best solution update is performed. At the end, the first solution of the solution list is returned on line 61 for later memory-reuse operations.

1.10 Experimental Results

Vertical Buffering

The implementation conditions are similar to what we considered in the last two chapters. As we discussed before, the expected time complexity of vertical buffering algorithm is $O\left(LB^2\right)$ where L is the number of wire sections and B is the number of available buffers in the buffer library. To study the effect of those two factors on the algorithm runtime, we run the vertical buffering

algorithm for different wire segments ranging from 1 to 1000 and for three libraries with a size of 2, 3 and 6. We purposely use the same buffers for all buffer libraries to have more meaningful runtimes. As it is shown in Table 4-1 and also in Figure 4-38, the relationship between the wire segment number and the runtime is linear. As the highlighted row demonstrates in Table 4-1, the impact of the buffer library size on the algorithm runtime is approximately quadratic, which agrees with the calculated time complexity $O\left(LB^2\right)$.

Table 0-1 Vertical Buffering Runtime for different wire segment numbers

Wire Segment (*100)	Runtime (seconds)		
	Buffer Library 1	Buffer Library 2	Buffer Library 3
1	0.078	0.094	0.359
5	0.328	0.563	1.844
10	0.734	1.156	3.549
15	1.031	1.657	5.235
20	1.344	2.157	8.203
25	1.609	2.625	8.563
50	3.203	5.156	17.922
75	5.484	7.719	24.844
100	6.438	11.250	33.250
150	9.250	15.140	49.344
200	12.485	20.765	70.735
300	18.515	30.328	103.234
400	25.484	40.532	136.500
500	31.813	53.453	169.063
1000	62.844	104.844	342.343

Table 0-2 Buffer libraries

Buffer Library	Buffer ID	Type	Intrinsic Delay ps	Output Resistance Ω	Input Capacitance fF
A	1	Inverting	2	15	15
	2	Non-inverting	20	4	3
B	1	Inverting	2	15	15
	2	Non-inverting	20	4	3
	3	Inverting	4	12	1
C	1	Inverting	2	15	15
	2	Non-inverting	20	4	3
	3	Non-inverting	4	6	8
	4	Inverting	2	4	5
	5	Inverting	4	12	1
	6	Inverting	10	10	10

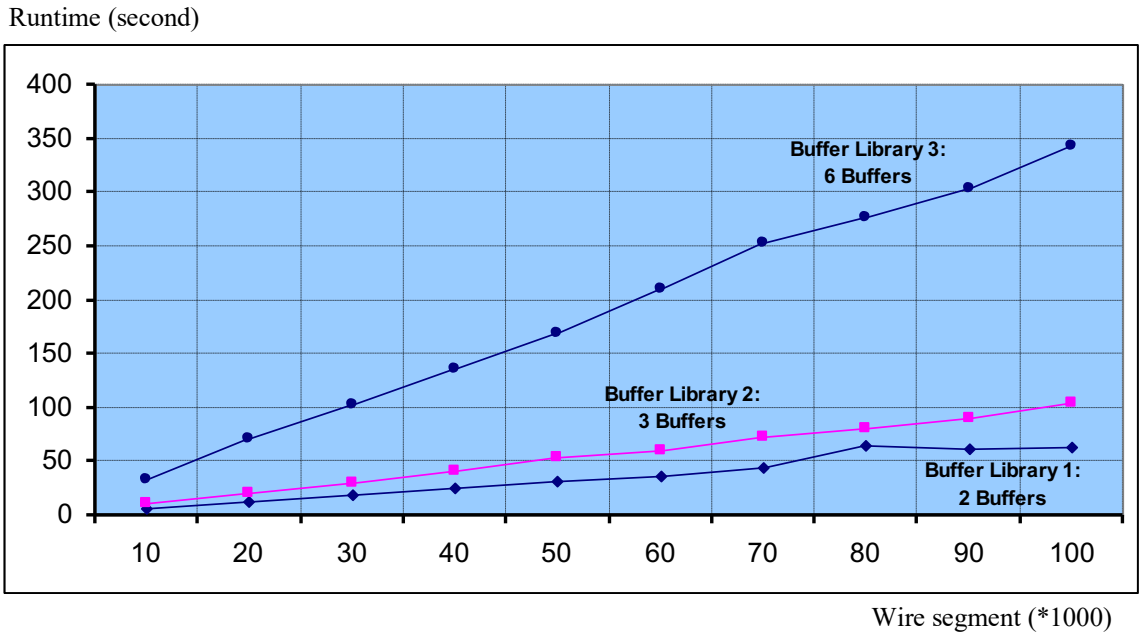


Figure 0-38 Vertical Buffering runtime for different cases

Main Algorithm Runtime

As we mentioned earlier, the emphasis has been on method adaptation rather than algorithm performance. As no effective bound is calculated to help searching within a reasonably small search space, we observe large runtimes for small fanout numbers, as shown in Table 4-3.

Table 0-3 Main algorithm runtime

Fanout	Runtime (seconds)
6	0.110
8	0.265
10	0.718
12	1.766
14	4.594
16	4.297
18	24.047
20	56.922
22	132.781
24	296.657
26	674.357
28	1536.235
30	3468.047

1.11 Conclusion

We have shown how one can effectively tackle complicated buffering problems in which the interconnect delay effect is also taken into consideration. We have proposed a number of models and methods to achieve such extensions on our balanced buffering algorithm. In order to preserve the solution quality, we have introduced two recursive functions to produce a complete feasible region of all possible buffer trees. A dynamic programming approach has also been presented to reduce the search space size by an early buffer insertion on the buffering solutions at the time of solution list creation. The memory-reuse and speedup concepts have also been modified such that we can profit from the existence of common subproblems to save time and memory.

Nevertheless, due to the lack of an efficient bound required to prune the search space, the algorithm does not present a good runtime. The issue of mathematically finding such a bound is beyond the scope of our Master's project and hence is left for prospective studies in this domain.

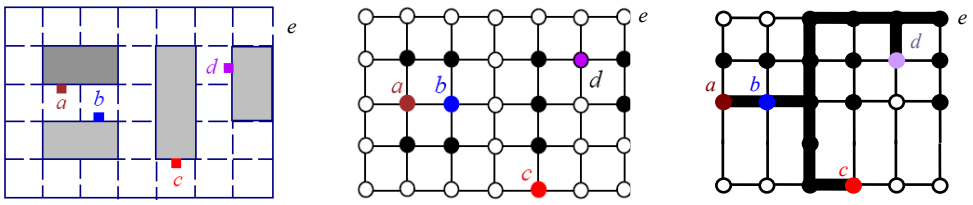
APPENDIX

P-Tree Method

The P-tree method or "Permutation-Constrained Routing Tree" was introduced by [CHENG ET AL. 96]. Although their algorithm does not take buffer insertion into account, it has been used in several buffering applications, and for that reason we briefly review it. [CHENG ET AL. 96] integrated wire sizing and power minimization with tree construction to solve the problem of maximizing the required time at the driver with loads, required times and positions taken into account. Their approach consists of two major phases:

I. Finding a proper order for the sinks heuristically
II. Generating the routing structure based on the order.

In the second phase, given an order for the sink nodes, the P-tree algorithm finds the optimal embedding of the net into the *Hanan grid*[1] , using a dynamic programming approach. Hanan grid is usually used to model the routing area. Each edge in the graph represents part of a routing track that can accommodate exactly one wire segment. Edges occupied by other nets and wiring obstacles are deleted from the graph. An example of how to use the Hanan grid for routing is shown in the figure below.



(1) (2) (3)

Figure i An output of P-tree method with Hanan grid for "*dbac*' order

One potential weakness of P-Tree method is that it is oblivious to sink criticality. Since the sink permutation is determined by the relative *physical locality* of the sinks, some very high performance and/or cost effective solutions may fall outside the solution space. For a delay

[1] The **Hanan grid** of a net is defined as the grid graph formed by the intersection of horizontal and vertical lines running through the terminals of the net.

optimization using buffering, it may be preferable to disregard (to a limited degree) the physical locality of such sinks at the expense of additional wire length.