UNIVERSITÉ DE SHERBROOKE

Université de Sherbrooke

Faculté de génie

Département de génie électrique et génie informatique

CIRCUIT DELAY OPTIMIZATION BY BUFFERING THE LOGIC GATES

OPTIMISATION TEMPORELLE DE CIRCUITS LOGIQUES PAR L'UTILISATION DE TAMPONS

Mémoire de maîtrise es sciences appliquées
Spécialité : génie électrique

Amir Hossein RABBANI

Sherbrooke (Québec), Canada Septembre 2007

Circuit delay optimization by buffering the logic gates

PREFACE

As devices shrink, deep submicron designs demonstrate the increasing importance of interconnect delay on the circuit performance. In order to reduce interconnect delay and help driving large fanout, buffer insertion needs to be performed during logic and physical synthesis. This optimization activity is often based on dynamic programming. In this dissertation, using the branch-and-bound technique, the problem for the specific case of buffering balanced trees is solved, where all loads have identical required time and input load capacitance. Necessary mathematical and data structural elements are provided to take into account a variety of design issues such as topology, buffer library and phase-shifting in the presence of inverting buffers. Combining dynamic programming and branch-and-bound techniques, a hybrid method is presented to improve runtime while memory consumption remains reasonably low. The underlying mathematical and algorithmic concepts given in this thesis can be used to generalize the proposed buffering method to produce a buffer tree for a set of different loads with different required time and capacitance.

Optimisation temporelle de circuits logiques par l'utilisation de tampons

RÉSUMÉ

Avec la miniaturisation actuelle, les circuits démontrent de plus en plus l'importance des délais d'interconnexion. Afin de réduire ce délai, l'insertion de tampons doit être effectuée durant la synthèse logique et la synthèse physique. Cette activité d'optimisation est souvent basée sur la programmation dynamique. Dans ce mémoire, la technique *branch-and-bound* est utilisé et le problème pour le cas spécifique d'arbres de tampons équilibrés est résolu, où toutes les charges ont un temps requis et une capacité identique. Une analyse mathématique est faite pour tenir compte d'une variété de questions de conception telles que la topologie, la bibliothèque de tampons et le changement de phase en présence d'inverseur. En combinant la programmation dynamique et les techniques branch-and-bound, une méthode hybride est présentée qui améliore le temps d'exécution tout en conservant une utilisation de mémoire raisonnable. Les concepts mathématiques et algorithmiques fondamentaux utilisés dans ce mémoire peuvent être employés pour généraliser la méthode proposée pour un ensemble de charges avec des capacités et des temps requis différents.

ACKNOWLEDGEMENTS

The author wishes to thank several people. I would like to thank Dr. Mailhot for his help, expert insight and valuable guidance which made this thesis an enjoyable experience. I would also like to thank Ali Shanian whose support and true friendship helped me a lot through this Master's project. Last but not least, I would like to thank all my friends, in particular Behnam Mostajeran and Zohreh Rafiei, for being beside me in difficulties and happiness.

To my parents:

Bijan Rabbani

Shahla Kamuei

For their unending love, boundless patience and supreme support

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Background and Motivation

In modern integrated circuits a logic gate often has to drive very large fanouts. This is due to the fact that during logic synthesis, where generating common expressions is needed to compact the circuit by reducing the number of logic gates, large fanouts are also produced. This is a crucial problem because a logic gate driving large fanouts dramatically slows down the circuit. On the other hand, there are many places where timing is so crucial that one has to make sure that the signals are traveling the circuit as quickly as possible, such as making clock trees, multiplexers, etc. Buffering is one common solution to these problems and is addressed during this dissertation. A new buffering method has been proposed and its efficiency is proved through a number of well-staged testing scenarios. The goal of the presented method is to construct the fastest buffer tree, i.e. a buffer tree by which signals can travel from a logic gate to its fanouts as quickly as possible.

A number of buffering methods have been suggested in the literature (see chapter 2). While these methods are efficient in terms of runtime and memory, none of them guarantees optimality. One often has to compromise runtime and memory consumption at the cost of solution quality. In this dissertation, a buffering method is proposed which produces the optimum solution for a specific class of buffering problems where all fanouts are identical in terms of input capacitance and required arrival time. The presented algorithm is significantly fast and needs very small memory. In constructing the buffer tree, no routing constraint is imposed on the tree structure. This is mainly because there are many applications where timing objectives dominate the geometrical objectives (such as designing a clock tree). In chapter 3, it is shown how one can take advantage of the special structure of this class of buffering problems to nicely model and solve them. The properties of the problem allow using branch-and-bound algorithm, which guarantees the optimality by nature. Other than optimality, one has to make sure that the solution is produced quickly and the calculations are done with a reasonable amount of memory. Therefore, a number of optimization techniques have been introduced in chapter 4 to improve the performance of the buffering algorithm proposed in chapter 3. It is going to be discussed how the solution space being explored by the buffering algorithm is characterized in such a

way that one can avoid redundant calculations by storing and retrieving common solutions. This optimization technique results in faster runtime. Some modifications on the branch-and-bound algorithm have been also put together to speed-up the buffer tree construction. In industrial scale problems, where synthesis tools have to deal with logic gates having hundreds of thousands of fanouts, memory consumption becomes a bottleneck. To avoid any memory overload in real applications, a new group of binary trees have been designed in this dissertation to help memory consumption remain within a reasonable range. A unique feature of those binary trees is the ability to modify the tree structure based on the way it is used, i.e. providing faster access times for the nodes used more frequently. This new class of binary trees, its properties, implementation and applications is going to be discussed more in details in chapter 4.

1.2 Contributions of the Thesis

In this thesis, a new balanced buffering method along with a number of speed-up techniques has been developed to address the buffering problem from a different perspective. The main contributions are listed as follows:

1- *An efficient way of balanced buffering using branch-and-bound algorithm:* While most of today's buffering techniques are based on dynamic programming, a new and effective method is proposed based on the branch-and-bound technique. Characterizing the problem from a mathematical point of view, the problem for the specific case of buffering balanced trees is solved, where all loads have identical required time and input load capacitance. This new method is called *balanced buffering.* As opposed to many buffering methods, the presented balanced buffering algorithm guarantees solution optimality and can handle several design issues simultaneously, such as buffer tree topology, phase shifting in the presence of inverters and buffer library. Also, the underlying concepts are provided for a generalized version of the proposed algorithm to solve the buffering problem for a set of different load capacitances and required times.

2- *Speed up techniques for the proposed balanced buffering algorithm:* To obtain faster search time for the proposed balanced buffering technique three efficient approaches are introduced for the first time in this thesis: mixed branch-and-bound and dynamic programming (or simply *mixed method*), *smart bound* and *solution jumper*. Applying these speed-up techniques, runtimes up to 60,000 times faster have been achieved for the simple balanced buffering

algorithm, which is a significant contribution to the CPU cost of the buffer tree construction method. While mixed method and smart bound are introduced to specially improve the runtime of balanced buffering, they can be applied to any similar problem where the efficiency of the branch-and-bound algorithm is affected by the existence of common sub-problems in the search space.

3- *Self-reorganizing binary search trees:* One of the speed-up techniques, the mixed method, needs to keep the results of solving each sub-problem in order to prevent redundant calculations during search traversal. Such approach requires a simple yet efficient structure to maintain and access the solution lists information in the memory. Lazy weight and perfectly balanced binary search tree, two new classes of dynamic search trees, are proposed for the first time in this thesis to improve the access time of those solution lists. Keeping record of each node usage number, they restructure themselves such that the nodes with higher access number move toward the root of the binary tree. This provides faster access time for highly shared sub-problems which generally improves the balanced buffering algorithm runtime.

1.3 Organization

The thesis is organized as follows. After a short introduction to the work addressed in this dissertation in chapter 1, basic buffering concepts, circuit synthesis stages and previous work in this field are introduced in chapter 2. Balanced buffering algorithm and its analysis are presented in chapter 3. Chapter 4 discusses the speed-up methods to improve the runtime of balanced buffering. Finally, a summary of current work and possible directions of future work is given in chapter 5.

2 OVERVIEW OF SYNTHESIS PROCESS & PREVIOUS WORK

In this chapter basic buffering concepts are introduced and a number of widely used buffer insertion techniques are presented. In order to understand how and where buffer tree construction is applied in the circuit design process, in section 2.1 different stages of circuit synthesis are explained and buffer tree construction is addresses at logic and physical synthesis stages. In section 2.2, major buffer insertion techniques are studied. The applications of these techniques encompass area and delay optimization in general, and physical effects such as wire capacitance, wire inductance and the effect of placement and routing in particular.

However, it should be pointed out that the buffering method proposed in this dissertation is new and is not an extension of any previously presented buffering method. Yet, reviewing these methods provides insight necessary for understanding the work that has been done in this dissertation.

2.1 Review of Buffering Concepts

2.1.1 What is a buffer?

A buffer is an amplifying element placed along the wires and between the logic blocks to help decoupling large loads and regenerating degraded signals. Though a buffer is conventionally known as a neutral logic gate that has no effect on the logic values it transmits, its definition in the domain of circuit synthesis has been extended to also include the inverters. In fact, a non-inverting buffer is a logic-gate compound of 2 contiguous inverters. As the most basic logic unit, inverters are generally smaller and faster than non-inverting buffers. However, the problem of phase-shifting in the presence of inverters makes many buffering algorithms only use non-inverting buffers. Nevertheless, solutions to a buffering problem can still contain inverters as long as the problem of phase-shifting is correctly handled. The symbolic forms of inverting and non-inverting buffers along with their circuits and logic function are shown in figure 2-1.

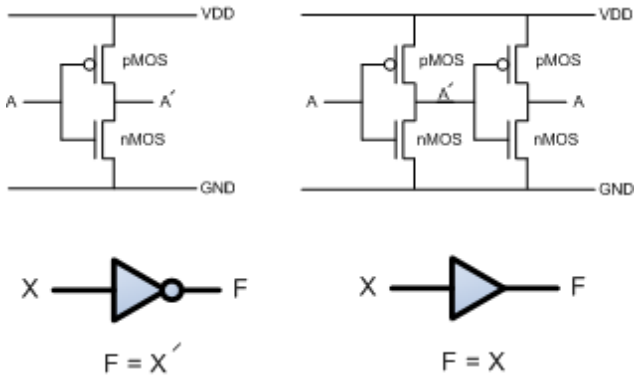


Figure 2-1 Inverting and non-inverting buffer

The physical and geometrical properties of buffers are specifically designed to yield fast timing characteristics while causing the least congestion at the routing level. Since different buffer layouts can result in different electrical properties, a number of inverting and non-inverting buffers with various physical characteristics are often put together in a library, in order to help automated design systems produce better buffering solutions. In figure 2-2, it is seen how 2 inverters with different layouts can present different physical characteristics. As it is shown in this figure, buffer layout 1 has small total capacitance, whereas buffer layout 2 has small total resistance and intrinsic delay.

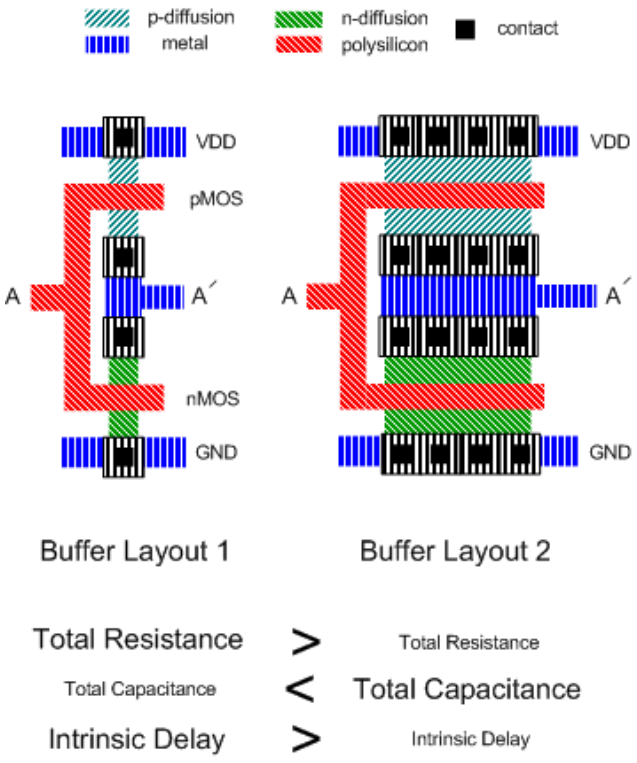


Figure 2-2 Two different implementations for an inverter

A buffer library provides the physical and geometrical diversity required in constructing a buffer tree. This diversity helps choosing the best buffer configuration for the physical characteristics of the given buffering problem. As a consequence of different layout design, a buffer [ALPERT et al, 2000]:

1) May have a relatively high delay when driving a small load, but a relatively low delay when driving a large load. Such a buffer usually is not a single gate, but rather a series of cascading buffers.
2) May be relatively fast for a large range of loads, but it may have a high input capacitance which increases the delay of the previous stage. This is typically true of large inverting buffers.
3) May have a low input capacitance, which is useful for decoupling a sub-tree that connects non-critical sinks, but the buffer may not have enough strength to drive the entire load that it needs to decouple.
4) May be designed for high noise margins or low power, but perhaps not the best performance.

Hence, many buffering algorithms are designed based on buffer selection techniques. Some of them will be introduce in section 2.2.

2.1.2 How to model a buffering problem

Buffers are typically inserted along the wires and at certain predefined places called *legal positions*. Legal positions are reserved empty blocks that are specified during the physical synthesis stage (physical synthesis will be introduced in section 2.1.3). Putting a series of buffers at these legal positions forms a *buffer tree*. An example of buffer tree is shown in figure 2-3. In this figure legal positions are shown by dashed triangles placed along a given routing topology between a source gate and 10 sinks (fanouts). To construct the best buffer tree, one has to decide which empty blocks must be filled out by what type of buffer. This also means that some legal positions can be left empty in order to make the best buffer tree. The number of legal positions in a given routing topology can vary with the physical design restrictions. This problem has been addressed by a number of researches that will be discussed in section 2.2.

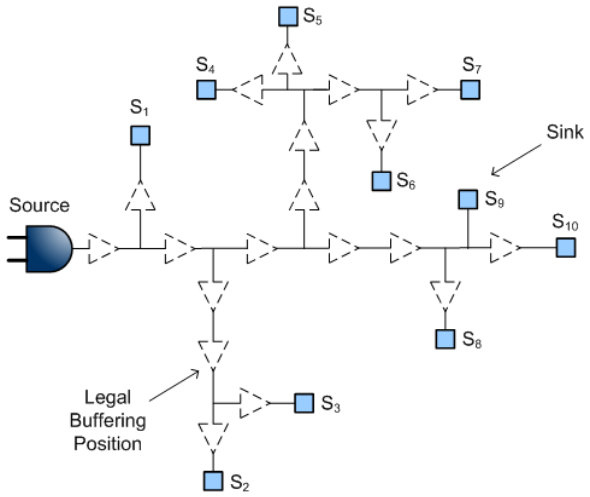


Figure 2-3 Modeling the buffer insertion problem

Different levels of circuit design process shall be discussed now in order to understand when and how buffer insertion is used in circuit delay optimization procedure.

2.1.3 Circuit Synthesis Stages

In the mid 80's, several academic and commercial systems (Such as *BooleDozer* by IBM and *Encounter RTL Compiler* by Cadence) were put together to address the problem of designing ever more complex digital circuits. In the early 90's, when those systems matured and were widely used, their focus broadened from initial optimization of area and delay to encompass power dissipation, testability and physical effects such as wire capacitance, wire inductance, and the effect of placement and routing on delay. Several automatic systems have been developed to help designing faster circuits and make optimization methods more efficient. These automatic systems are all based on 3 main levels, as illustrated in figure 2-4. In *High Level Synthesis*, the circuit functionality and the input-output behavior is designed using a high-level hardware description language (HDL). This functionality is then transformed to a netlist in *Logic Synthesis* followed by technology mapping. In *Physical Synthesis*, the netlist is transformed into networks of transistors and interconnects and then it is fabricated. While devices shrink in size, meeting the area, time and power consumption objectives becomes more and more critical. As a result, one has to iterate between these three synthesis levels until the cost and speed targets are met.

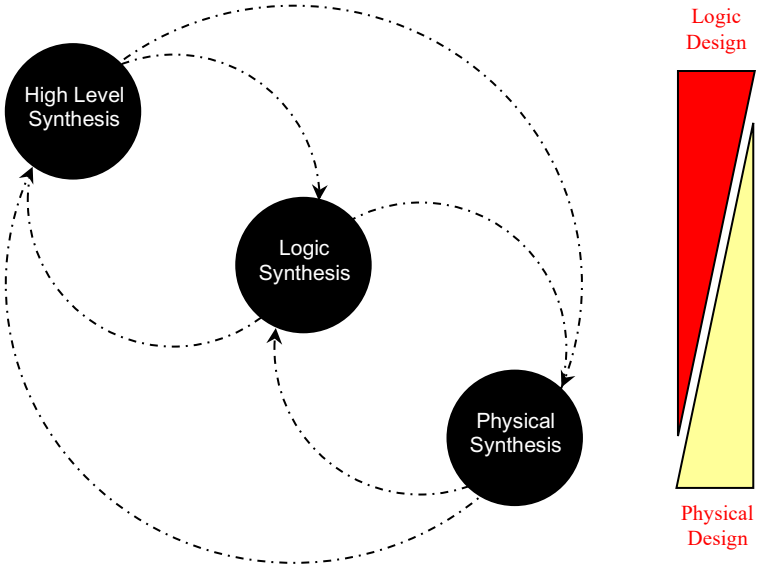


Figure 2-4 Circuit synthesis stages

2.1.4 Different levels of buffering

As legal positions for buffer insertion are specified in physical synthesis, buffering was originally considered as a post-layout optimization method. However, due to the importance of timing requirements, in modern circuit synthesis tools buffering is done during logic synthesis and then is re-optimized together with physical synthesis. In addition, there are some considerations during high-level synthesis to make buffering a more effective optimization technique in logic and physical synthesis. The situation of buffering in the circuit design sequences is illustrated in figure 2-5. During logic synthesis and when timing properties of the circuit are found, buffer insertion is applied to improve the circuit speed (black box in figure 2-5). Then during placement and routing steps the buffering solution provided by logic synthesis is re-optimized (grey boxes in figure 2-5). This is done at the local placement and detailed routing steps. A number of iterations between high-level, logic and physical take place until the best solution is found.

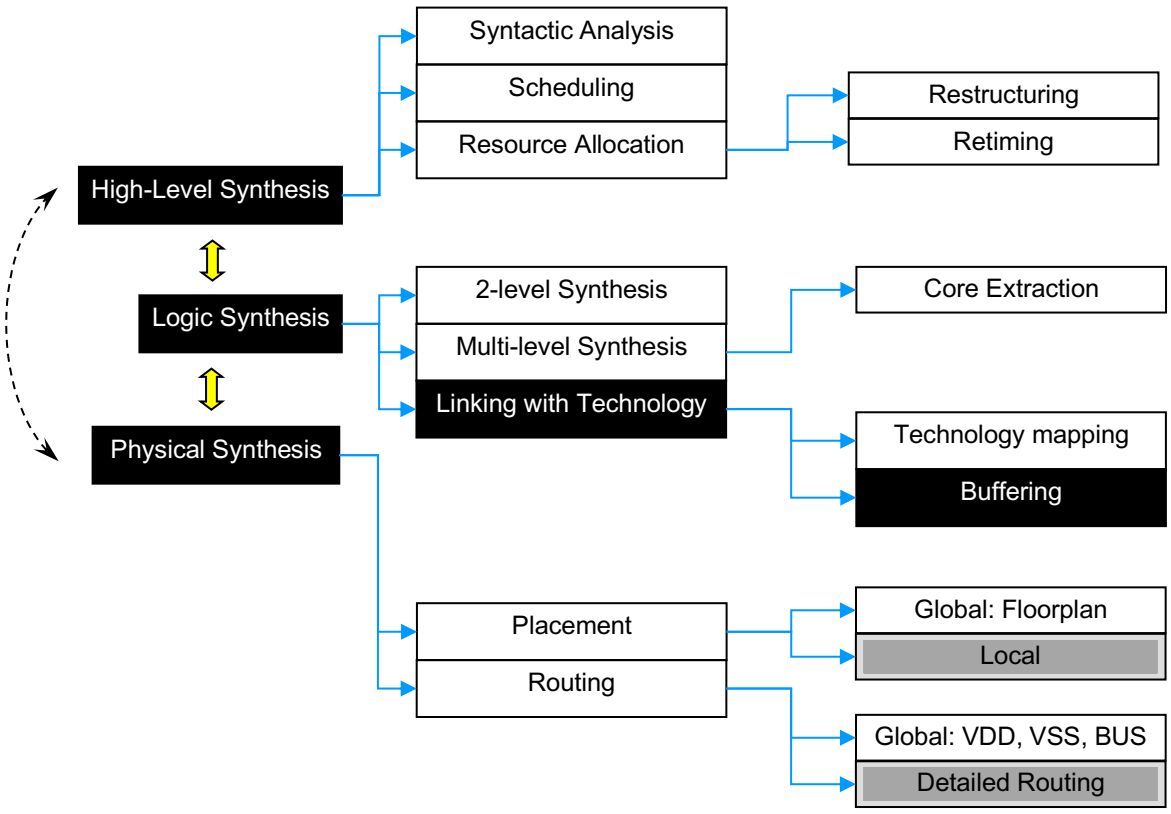


Figure 2-5 Buffering in the circuit synthesis steps.
Grey boxes indicate buffering considerations during physical synthesis [MAILHOT, 2005]

During the next two sub-sections some optimization problems in logic and physical synthesis will be examined where buffering techniques play a major role in resolving them.

2.1.5 Buffering in logic synthesis

Using an RTL (Register Transfer Language) description as an entry, logic synthesis generates a structural view of a logic-level model and converts this data structure into a network of generic logic cells, called **netlist**. The synthesis process consists of a sequence of optimization steps, the order and nature of which depend on the chosen cost function-area, speed, power, testability, or some combination. Typically, logic optimization systems divide the task into the following steps [SMITH, 1998]:

1) A technology independent phase, where the input RTL is parsed (also called analysis) and translated (also called elaboration) to a data structure. This data structure is converted into a network of generic logic cells.
2) Next is the logic optimization phase where the logic is optimized using a number of Boolean or algebraic manipulation techniques. Many optimization algorithms can be employed here based on the logic involved (combinational or sequential). This step attempts to improve this technology-independent network under the control of the designer. The output of this stage is an optimized, but technology-independent logic network.
3) Further is technology-mapping (also called logic-mapping) phase, where the synthesizer maps the optimized logic to a specified technology-dependent target cell library. This phase takes into account the properties of the intended implementation architectures. The technology-independent description resulting from the earlier phases is thus translated into a gate netlist.

A schematic logic synthesis process is shown in figure 2-6.

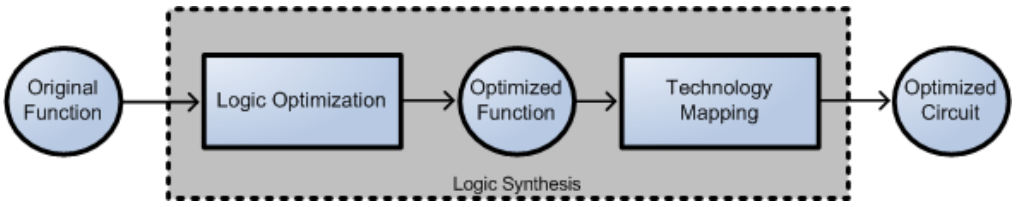


Figure 2-6 Logic synthesis stages

We now explain a very common but crucial problem that must always be handled in logic synthesis, and it will be discussed how and at which level buffer insertion can help the situation. Some basic concepts [CARRAGHER and CHENG, 1995] will be defined that are important in the rest of this thesis:

Definition 1. *Arrival Time*: the actual arrival time a_g of gate g is the latest time for which a valid signal will be produced by g for its fanouts.

Definition 2. *Required Time*: the required time r_g of gate g is the latest time for which a valid signal is needed on g's fanouts.

Definition 3. *Slack*: the slack of gate g, s_g, is the difference between its required time and its arrival time, or $s_g = r_g - a_g$. The slack of a gate represents how well the timing

requirements are being met at that gate, if its slack is positive, and how poorly those requirements are not being met at that gate, if the slack is negative.

Definition 4. *Critical Path*: path of smallest slack (usually negative) from primary inputs to primary outputs.

During certain logic synthesis steps, some gates with very large fanout are produced, as a result of sharing logic functions. A gate which has to drive many others can significantly slow down the whole circuit, if it is located on the critical path. Fanout optimization in logic synthesis addresses the problem of distributing an electrical signal to a set of sinks with known loads and required times so as to maximize the required time at the signal driver (root of the net). Interconnect delay is not incorporated in this operation because the locations of sinks are not known at this stage. There are two main methods to break large fanouts into smaller portions: splitting and buffer insertion.

One solution to the fanout problem is to split the fanouts of a gate into several parts, each of which driven by a copy of the original gate. To this end, gate duplication is applied to maintain the main logic function while helping a fair load distribution. An example of splitting fanouts by duplicating a logic gate is shown in figure 2-7.

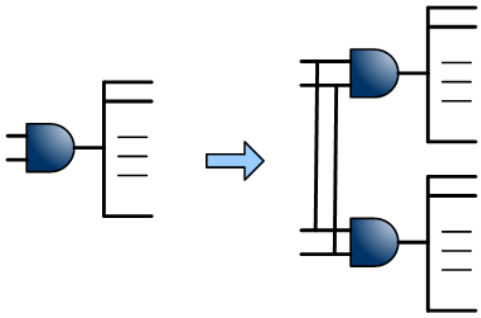


Figure 2-7 Splitting the fanouts of a gate into several parts.
Each part is driven with a copy of the original gate.

Although gate splitting speeds up the gate being split by reducing the output load, it increases the load of the gate driving the split gate. This operation is therefore beneficial at times, but not always.

Another technique used for fanout optimization is buffer tree construction. Buffers are logic gates exclusively designed for driving signals applied to load capacitances and optimizing the signal's arrival time. In effect, a buffer can hide a fanout with large load from the other fanouts with smaller loads, thus reducing the delay for the driving gate. After having mapped the network during the technology mapping step of logic synthesis, delay estimation techniques are used to determine whether

the delay requirements are met, and whether maximum loads of logic gates are violated. Then a primary buffering is done in order to get realistic delay evaluation. After obtaining more realistic delays from physical synthesis, the buffer trees is re-optimized, iterating between logic and physical synthesis. Critical paths are also detected during logic synthesis, so buffering at this moment decouples large loads off of these critical paths. In figure 2-8, buffer insertion that takes place after technology independent optimization is highlighted in the process of circuit optimization during logic synthesis.

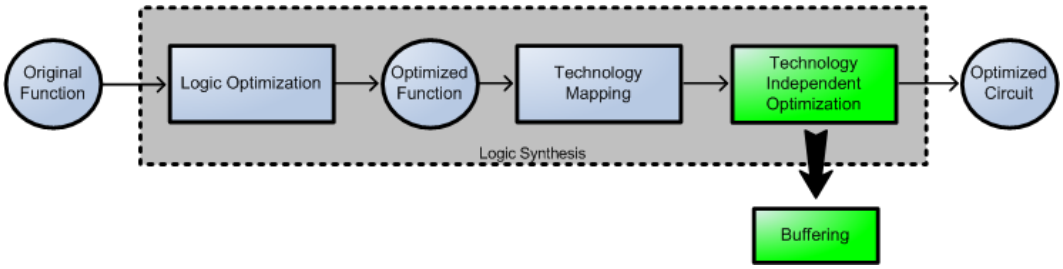


Figure 2-8 Buffering during logic synthesis

Most often buffer insertion is more effective than gate duplication since buffers are usually better designed for driving signals. Also, gate duplication can increase routing congestion and make placement more difficult.

The two main fanout optimization techniques involving buffering synthesis are *critical path isolation* and *balanced load decomposition*.

1) **Critical Path Isolation:** During logic synthesis, when one or several sinks (fanouts) are timing-critical, the critical path isolation technique generates a fanout tree so that the root gate drives the critical sinks while the non-critical sinks are separately driven by a buffer tree. In figure 2-9 an example of an optimized circuit with 3 buffers shows how the fanouts of a logic gate are divided into critical and non-critical parts such that non-critical fanouts can be driven with buffers.

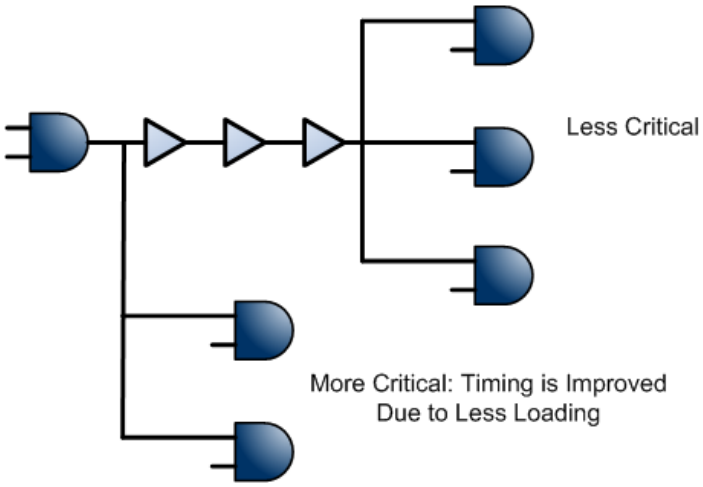


Figure 2-9 Dividing less critical fanouts with buffers

2) **Balanced Load Decomposition:** If sinks required times are within a small range, balanced load decomposition is applied in order to decrease the load at the source gate, using buffer insertion. This is shown in figure 2-10.

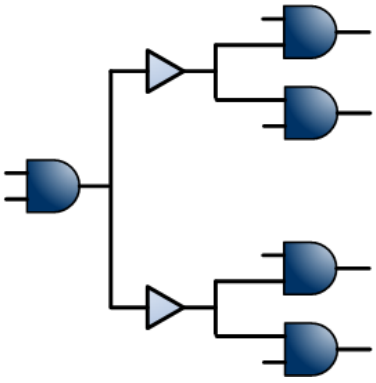


Figure 2-10 Balanced Load Decomposition.

2.1.6 Buffering in physical synthesis

Due to the importance of ever challenging problems of interconnect delay (wire delay) and its impact on physical design, first the interconnect delay problem and its potential solutions are discussed, and then the way that buffering can contribute to solving this problem is examined.

As feature size becomes smaller 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. Starting with the 0.25 μm generation, circuit delay has been dominated by interconnect delay, as it is explicitly shown in figure 2-11.

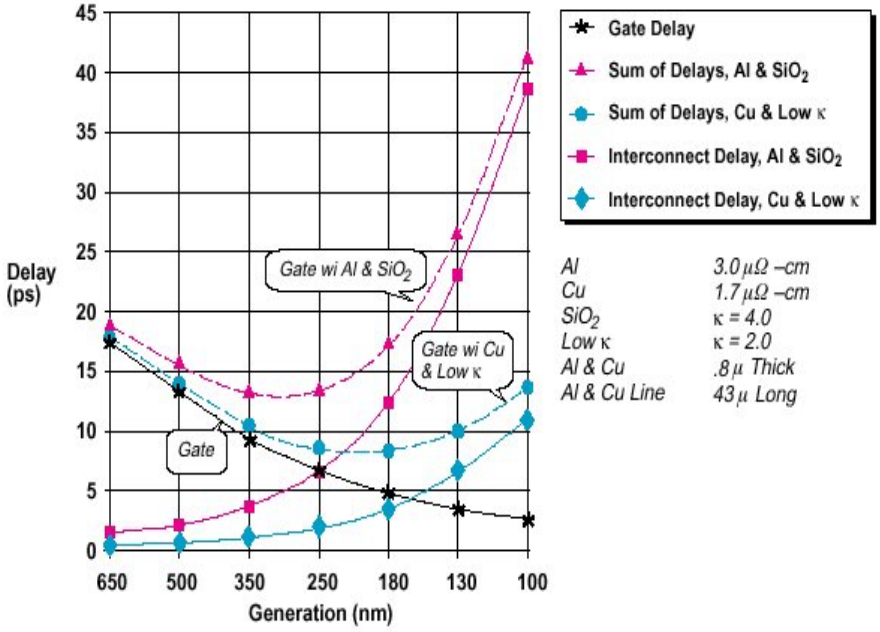


Figure 2-11 The dominance of interconnect delay.
Calculated gate and interconnect delay versus technology generation illustrating the dominance of interconnect delay as feature sizes approach 100 nm. [BOHR, 1995]

Vast efforts have been taken to control interconnect delay. There exist two main techniques:

Processing technology: New materials, such as copper and low dielectric constant (K) materials, have been used to improve interconnect performance. However, at the global interconnect level the benefit of material changes alone is insufficient to meet overall performance requirements. Even with the help of copper and low (K) materials, it is predicted that interconnect delay is still likely to dominate the chip performance beyond the 100nm technology. Therefore, the significance of interconnect delay is expected to rapidly increase in the near future.

Design Technology: Using automated design tools and efficient algorithms, a permanent goal is to reduce circuit delay during the synthesis steps. Miscellaneous interconnect performance optimization techniques are done in synthesis procedures, like topology construction, buffer insertion, driver sizing, wire sizing and wire spacing. It has already been explained how and at which level buffering is done in logic synthesis. The methods that take buffering concerns into account during physical design are briefly reviewed, starting with an introduction to physical synthesis.

Having generated netlists during logic synthesis, physical synthesis performs the operations needed to produce the final circuit. The necessary steps to electrically implement the initial circuit design are as follows:

1) **Partitioning**: if too large to fit into one ASIC (Application Specific Integrated Circuit), functional blocks are split, or partitioned, into smaller blocks considering predefined design objectives. The product of this phase is then a netlist describing circuit blocks.

2) **Floorplanning**: having a hierarchical netlist that describes the interconnection of the blocks, floorplanning tools map this netlist into a physical description. The task of floorplanning ranges from arranging logic cells within the blocks to deciding about I/O pads and type of clock distribution.

3) **Placement**: At this level, logic cells are placed within flexible blocks. Since, after floorplanning and placement both intrablock and interblock capacitances are predictable, it is possible to return to the logic synthesis domain to re-optimize the design with more accurate estimates of the capacitive loads that each logic gate must drive.

4) **Routing**: The routing task consists of making connections between the blocks and within the designated channels defined by earlier phases.

5) **Compaction**: To minimize the overall circuit area, a set of optimization actions are performed to make the routed blocks as compact as possible.

6) **Extraction and Verification**: During the extraction step, the interconnection resistance and capacitance are determined and together with the entire design are sent for timing verification. At this step, the cost and design objectives along with the logical functionality are tested to ensure all design targets are met.

A summary of the physical synthesis steps is provided in figure 2-12.

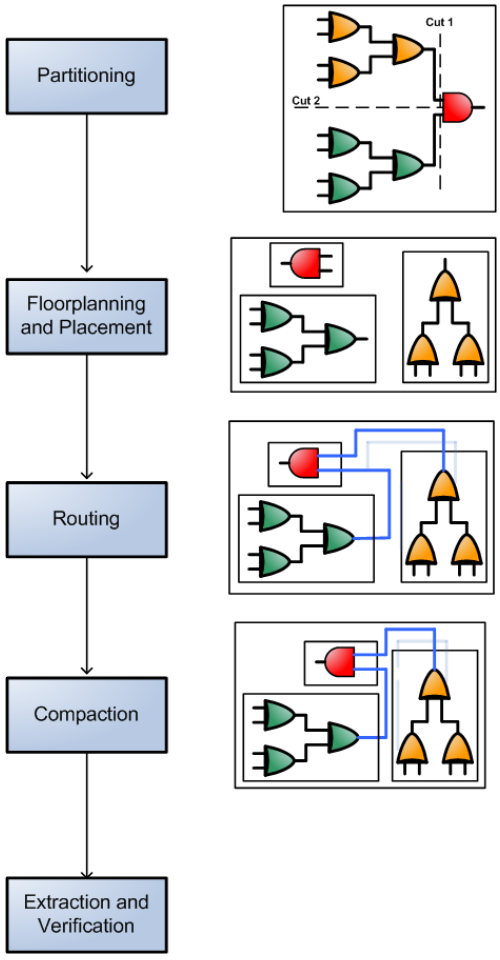


Figure 2-12 Physical synthesis procedure

[SMITH, 1998]

As mentioned before, buffering considerations are taken into account in physical synthesis. This is done at 3 different levels of physical synthesis [CONG and YUAN, 2000]:

1) **Pre-routing (Placement and Floorplanning) stage**: To obtain better routability, a number of methods consider some particular places for buffer insertion during the floorplanning step. This is helpful because during or after the routing step, most of the area is occupied by logic blocks and wires. Therefore ignoring buffer insertion during pre-routing steps can seriously restrict the space available for buffering.

2) **Routing stage**: In conventional design flows, fanout optimization and routing generation are often performed in a sequential manner, which means buffer insertion is used as a post-layout optimization technique after the routing stage. Consequently, a solution obtained during one of these optimizations becomes a constraint for the other one. Solving the unified problem, i.e. generating a buffered routing tree for a set of sinks and a driver, helps capturing the intrinsic interactions between the combined design steps and produces higher-quality implementations by systematically searching a much larger solution space.

3) **Post-routing stage**: Buffer insertion is typically a post-layout optimization technique, meaning that it is applied to improve the layout and delay after the routing stage. Having more realistic wire and load capacitance estimates at this level, the previously generated buffer tree is likely re-optimized to yield better circuit timing.

As devices shrink in size, the focus is being narrowed more and more on buffering in physical synthesis. In the next section, the major work done for buffer tree construction in both logic and physical synthesis will be briefly reviewed.

2.2 Previous Work

There are two main groups of buffer insertion techniques: Van Ginneken's method [VAN GINNEKEN, 1990] and its variations, and other methods which are not extensions of Van Ginneken's method. As many of the practical buffer insertion techniques in use today are based on the important work of Van Ginneken, this algorithm is studied first. Then other major efforts which basically extend Van Ginneken's method are examined. Buffer insertion methods that are not extensions of Van Ginneken's algorithm are discussed last. Different issues such as multi-type buffering, simultaneous routing and buffering, buffer sizing, continuous buffer insertion and buffered clock trees are addressed in this section which provides a good insight about the major challenges in this field.

2.2.1 Early efforts on optimal buffer insertion: Van Ginneken's algorithm

Van Ginneken proposed a dynamic programming algorithm [BELLMAN, 1957] for inserting buffers into a given topology. His algorithm returns the optimal solution in terms of Elmore delay [ELMORE, 1948], taking RC effects into account. For given required times at the sinks of the wiring tree, the algorithm chooses buffers such that the required time at the source is as late as possible. The topology of the wiring tree is assumed given, as well as the legal positions for the buffer insertion. The algorithm uses a depth first search on the wiring tree to construct a set of connected capacity-required time pairs (C, Q) that correspond to different choices for possible assignment of buffers. These pairs are inserted directly after branching points and at the legal positions. The structure of Van Ginneken's algorithm is shown in figure 2-13.

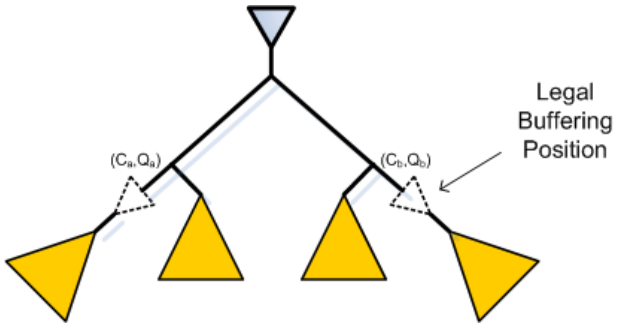


Figure 2-13 Van Ginneken's algorithm.

As the algortihm is based on dynamic programming, it consists of two main phases: bottom-up prediction and top-down decision making.

During the first phase the algorithm computes all options for each node. A set of (C, Q) pairs are constructed and stored. Then, for the options at the root of the tree delay, options are calculated and the option with the best source delay is chosen. The final solution is constructed during the second phase where the computations that led to the best option at the source are traced back. Buffers are placed during this phase.

According to Van Ginneken, in addition to timing optimization, the number of buffers can also be optimized. This is done by using triples of numbers rather than pairs for the options. Each option, in addition to the required time and the load, also has a solution cost. At the time of decision making, an option can only be discarded if it is worse in all three respects.

2.2.2 Extensions of Van Ginneken's algorithm

Despite its optimality under certain conditions, Van Ginneken's algorithm has some drawbacks as well. The time and space complexity of Van Ginneken's method is O (n^2) [ALPERT et al, 2000] [ZHOU et al, 2000] where *n* is the number of buffer positions. Therefore, for large fanouts this method becomes inefficient. Besides, Van Ginneken' algorithm only works with a single-type of non-inverting buffers where only one legal buffering position is considered between two nodes. A number of techniques have consequently been proposed to efficiently enhance the complexity of the algorithm [SHI and ZI, 2005] or to allow the algorithm to work with a buffer library consisting of inverting and non-inverting buffers with different physical characteristics [HRKIC and LILLIS, 2002] [SHI and ZI, 2005] [ALPERT et al, 2000].

One can improve the time and memory complexity of Van Ginneken's by performing a set of modifications on the original algorithm. These modifications are based on finding and removing redundant solutions and are performed through 3 steps [SHI and ZI, 2005]:

1) *Predictive Pruning*: Examining the options produced during the first phase of Van Ginneken's algorithm, one notices that some options are potentially dominated by some other options. In fact, whenever option A provides larger time slack and smaller input capacitance than option B, option A dominates option B, i.e. option B becomes a redundant solution. Speed-up is achieved by finding and pruning future redundant solutions.
2) *Making Option Tree*: Organizing the options information in an efficient data structure like a balanced binary tree helps achieving faster decision making process during the second phase of Van Ginneken's problem. Utilizing such a system also results in smaller memory consumption.
3) *Fast Merging*: Having one balanced binary tree for each option and its sub-tree information, the final solution is quickly constructed by merging those binary trees during the bottom-up phase.

Performing these techniques Van Ginneken's approach time complexity reduces to O ($n \log^2 n$) while only O ($n \log n$) memory is needed to construct the buffer tree.

To remove the drawback of single type buffering, some methods have been introduced to make it possible to do multiple buffer insertions. A primary solution quality improvement is achieved by taking *b* buffer types into account in the original Van Ginneken's algorithm. However, this basic

extension leads to $O(b^2n^2)$ runtime [HRKIC and LILLIS, 2002]. Better runtime is achievable if the 3 mentioned modifications consider a buffer library as well. This results in $O(b^2n \log^2(n))$ runtime [SHI and ZI, 2005]. Modern design libraries may contain hundreds of different buffers, which may be either inverting or non-inverting. If a user supplies every buffer available for the given technology as input to the buffer insertion tool, it could possibly take several days or even weeks to run to completion on a large design. Consequently, an appropriate set of buffers must be carefully selected to reduce the runtime [ALPERT et al, 2000]. This is done in two steps. First, according to the physical characteristics of buffers a pruning process is applied to find the *superior* buffers and discard the rest. These superior buffers are chosen based on a performance criteria defined by the user, such as intrinsic delay, high noise margins, etc. This subset of superior buffers may be larger than the allowable buffer library size, again defined by the user. Therefore, during the second step, similar buffers are clustered and smaller buffer libraries are formed. A new size-reduced buffer library is created by choosing a number of smaller libraries. The metric used for proximity between buffers is their timing properties expressed as a linear delay function.

A different solution to multiple-buffer insertion is buffer sizing. Instead of a discrete buffer library, some methods allow for continuous buffer sizing [VOGEL and WONG, 2006] [CHEN et al, 2002] [CHU and WONG, 1997]. As the input capacitance and the output resistance of the buffer can be expressed as the linear functions of the buffer size, different timing properties such as faster rise time or faster intrinsic delay are achieved by varying the transistor widths of the buffer. This removes the need to have a buffer library and also helps meeting the timing requirements of the circuit, but at the cost of more computational efforts.

Van Ginneken's algorithm assumed only one buffer per wire in the tree. This assumption can severely hurt solution quality if the wire delay is taken into account. Instead, one can divide each wire into smaller *segment*s, and hence introducing 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 wire segments 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 [ALPERT and DEVGAN, 1997] and the appropriate number of wire segments has been computed. This is done based on using only one buffer type. Handling a buffer library is achieved 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 [ALPERT et al, 2000].

The extensions of Van Ginneken's algorithm mentioned above are done mainly during logic synthesis. However, some other extensions take buffer insertion into account during physical synthesis. These extensions are categorized in 2 groups:

1) **Simultaneous routing and buffer tree construction (buffered routing)**

In modern circuit fabrications chips becomes more congested, the number of metal layers used increases and interconnect delay dominates gate delay in establishing the overall circuit performance. Consequently, the important tradeoff between routing resource cost and signal delay is unavoidable. Some researchers consider simultaneous routing tree construction and buffer insertion to tackle this problem [TANG and WONG, 2004] [OKAMOTO and CONG, 1996a] [LILLIS et al, 1996a] [SALEK et al, 1999] [CONG and YOUAN, 2000] [HRKIC and LILLIS, 2002], which is an NP-hard problem [SHI et al, 2004].

Early efforts on buffered routing started with combining routing techniques with Van Ginneken's algorithm in 1996. While a method [OKAMOTO and CONG, 1996b] was proposed to construct the fastest buffer tree based on the A-Tree routing topology [CONG et al, 1993] , some other approaches [LILLIS et al, 1996a] [OKAMOTO and CONG, 1996a] were proposed to use P-Tree routing topology [LILLIS et al, 1996b] during buffer insertion. In 1999, a more general buffered routing algorithm based on the P-Tree method was proposed [SALEK et al, 1999]. This method is called MERLIN. Introducing Cα-Tree as an extended version of P-Tree, MERLIN solves the buffered routing problem in polynomial time where multiple-buffer insertion is also allowed. A concept that MERLIN introduced for the first time was the 3-dimensional curves to take buffer location and area into account. These 3-dimensional curves consist of required time and load capacitance versus total buffer area. The third dimension (total buffer area) allows the user to solve the problem for either one of the following variants:

I) Minimizing the required time subject to an area constraint
II) Minimizing the area subject to a required time constraint

2) **Handling buffer insertion during floorplanning (pre-routing stage)**

As the amount of communication among modules rapidly increases, it becomes more and more difficult to insert buffers to remedy interconnect during or after routing,

since most silicon and routing resources are already occupied. To that effect, some solutions have been proposed to consider buffering before routing and during the floorplanning stage by reserving particular areas for buffers [JIANG and CHANG, 2004] [KANG et al, 1997a]. These reserved areas are known as *buffer blocks*. The *Buffer Blocks* are used to guarantee an effective interconnect optimization during the routing stage. The designated regions for buffer insertion may significantly change the floorplan and placement, thus causing problems in timing closure and design convergence. It is possible to do buffer block planning during the floorplanning stage [JIANG and CHANG, 2004], or construct a bounded delay tree, and then use Van Ginneken's algorithm to optimize buffers [KANG et al, 1997a].

A summary of the time complexity of Van Ginneken's approach and some of its variations is shown in table 2-1 (at the end of the chapter). In this table *n* represents the number of sinks and *b* represents the number of buffers available in the buffer library.

2.2.3 Other Work

Due to the importance and wide applications of buffered routing, numerous methods have been proposed that are not extensions of Van Ginneken's algorithm. Many of these methods are graph-based and are known as *maze routing* approaches [LAI and WONG, 2000] [ZHOU et al, 2000] [HUANG et al, 2003]. The goal of maze routing is to find a route between two terminals in a routing area, which is often represented as a grid graph. Some wiring obstacles and restrictions on buffer locations and types may be present in the routing area. One major advantage of maze routing approaches over the extensions of Van Ginneken's algorithm is that they are formulated as shortest path problems. Therefore, efficient software routines solving shortest path problems in existing graph application libraries can be used in buffered routing. A sample routing grid graph and a buffered minimum delay path is shown in figure 2-14 [LAI and WONG, 2000]. The dark areas represent wiring obstacles. Buffers can not be placed in gray and dark regions, while wires are allowed to pass through gray areas. Solid circles at some of the grid line intersections are possible buffer locations or previously mentioned legal positions.

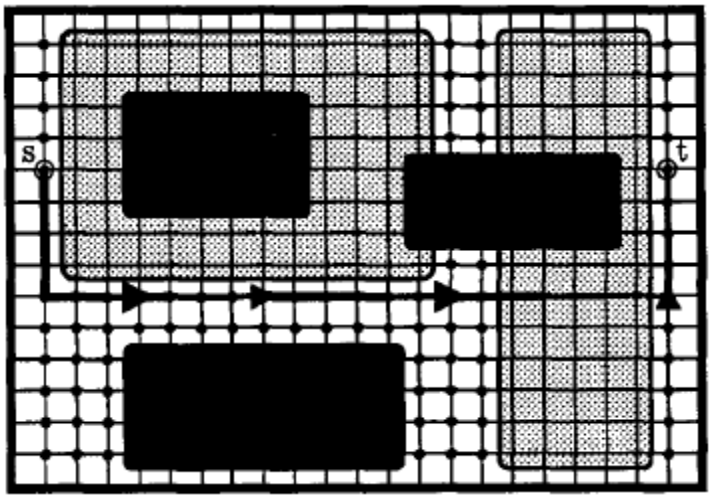


Figure 2-14 A routing grid graph and a buffered minimum Elmore delay path.

For a buffer insertion technique to be effective, it must be fully aware of its surrounding blockage constraints. During the routing process, there are macro-blocks placed within the area. These blocks form useful routing regions because wires are allowed to run over them, whereas buffers are not allowed to use them, since in that case the design of those blocks has to be changed. One brute force solution is to ignore macro blocks during routing. In this way, first a shortest path is found and then buffers are inserted outside the macro blocks. If there is no macro block, it can be proved that this sequential two-stage routing and buffer insertion approach gives an optimal solution. However, with macro blocks, a shortest path no longer guarantees minimum delay. The *fast path* algorithm [ZHOU et al, 2000] is a way of simultaneously doing routing and buffer insertion with blockage avoidance. This method extends Dijkstra's shortest path algorithm to do a general labeling, based on the Elmore delay model where path length is substituted by Elmore delay in Dijkstra's algorithm. Unlike Dijkstra's algorithm, the sub-path of a shortest path in this solution is not necessarily a shortest one. The total runtime of the fast path algorithm is O(nv(e+nv) log(nv)), where *n* represents the number of possible buffer positions, *v* is the number of vertices and *e* is the number of edges. A similar work that also considers wire sizing has a $O(v^2 \log(v))$ runtime and $O(b^2v^2)$ space complexity where *v* is the vertices in the grid graph and *b* is the number of buffers available in the buffer library [LAI and WONG, 2000]. In buffered routing more accurate delay models can also be used (transmission lines, delay look-up tables, etc.) [HUANG et al, 2003]. In this method, only those vertices which have qualifying transition time are included in the graph. This technique guarantees that all transition time constraints are satisfied. With *k* transition time bins and *v* vertices in the grid graph, the time and space complexity of this method is $O(k^2v^2)$.

In addition to maze routing algorithms, some studies have been done on different tradeoffs in buffered routing. As a result of the intrinsic complexity of buffered routing, one has to carefully deal with various design parameters where each design objective becomes a constraint for the other ones. Based on a previously proposed buffered routing method [TANG et al, 2001], [TANG and WONG, 2004] widely discusses different approaches to tackle the tradeoff between signal delay and routing cost by formulating the problem as a linear function of all design constraints.

As opposed to the buffering methods reviewed so far, a number of approaches assume no restriction for buffer positions and buffer sizes. These methods are known as *continuous methods*. For buffer insertion on a single line allowing continuous buffer positions and continuous buffer sizes, Dhar and Franklin [DHAR and FRANKLING, 1991] proposed a closed form solution, and Chu and Wong [CHU and WONG, 1999] proposed a quadratic programming approach. However, it should be pointed out that as opposed to the discrete version of the buffer insertion problem, the continuous methods can not be applied to trees. This drawback limits the applications of such approaches.

The minimization of total wire length is of interest since total wire length contributes to circuit area and routing congestion. As a result, some methods have been proposed to optimize total wire length as primary objective, with satisfying delay bounds as secondary objective [ZHU, 1995] [KANG et al, 1997b]. These methods are basically known as d*elay bounded* algorithms. *Delay bounded minimum Steiner tree* or DBMST is one way to construct a low cost Steiner tree with bounded delay at critical sinks [ZHU, 1995]. The DBMST algorithm consists of two phases:

(1) Initialization of Steiner tree subject to timing constraints
(2) Iterative refinement of the topology to reduce the wiring length while satisfying the delay bounds associated with critical sinks.

Since the Elmore delays at sinks are very sensitive to topology and they have to be recomputed every time the topology is changed, the DBMST algorithm searches all possible topological updates exhaustively at each iteration and as a result is very time consuming. The *delay bounded minimum buffered tree* or DBMB-tree algorithm as an extension of DBMST has smaller time complexity $O(n^2)$ where n is the total numbers of the terminals of the net [KANG et al, 1997b]. This algorithm successfully combines the local stochastic hill climbing features from SA (Simulated Annealing) and the global crossover operation from GA (Genetic Algorithm) in an optimization method, named genetic simulated annealing (GSA). Also, a multi-dimensional acceptance function is defined to accept the candidate solutions along the single search path generated by SA-based local moves. This multi-

dimensional function is defined based on the votes of the experts, and the objectives are ordered by sensitivity defined for each of them.

More accurate delay models have been investigated by some researchers. While *RC* (Resistance-Capacitance) models are used for high resistance nets, inductance is becoming more important with faster on-chip rise times and longer wire lengths. Wide wires are frequently encountered in clock distribution networks and in upper metal layers. These wires have low resistance and can exhibit significant inductive effects. Furthermore, performance requirements are pushing the introduction of new materials for low resistivity interconnects. Inductance is therefore becoming an essential element in VLSI design methodologies. The Elmore delay does not consider non-monotonic responses which can occur in *RLC* (Resistance-Inductance-Capacitance) circuits. Therefore, some approaches have been proposed to take wire inductance into account. For example, one method introduces a simple tractable delay formula for *RLC* trees that preserves the useful characteristics of the Elmore delay model [ISMAIL et al, 1999]. In this method the rise time of the signals in an *RLC* tree is characterized as well as the overshoots and the settling time.

Clock tree buffering is also addressed by a number of methods [WANG et al, 2005] [TELLEZ and SARRAFZADEH, 1997] [ALPERT et al, 2001]. In high performance synchronous VLSI design, system performance is limited by the quality of its clock signal, which is measured by the *clock skew*, *clock slew* and *clock phase delay*. The clock skew is defined as the maximum difference between the arrival times of the signals at all of the clock sinks. The clock slew is the slope of clock signals and the clock phase delay is defined as the maximum delay from the clock source to any clock sink [WANG et al, 2005]. Without a careful design, clock skews can cause lower clock frequency (zero clocking) as well as race conditions that result in failure regardless of frequency (double clocking). These two important factors can be optimized by good routing strategy and effective buffer insertion. The most common clock distribution network is a buffered tree. The focus of buffered clock tree systems usually is on skew minimization whereas a good buffer tree can also improve slew rate. Since bounding the load capacitance is a well known method to improve coupling noise immunity most of the buffered clock-tree algorithms use bounded load capacitances to guarantee meeting the electrical constraints [TELLEZ and SARRAFZADEH, 1997] [ALPERT et al, 2001]. However, one major drawback of such algorithms is that they can only use single non-inverting buffer type.

2.2.4 Summary

Major approaches applied in logic and physical synthesis to handle buffering in different contexts with different objectives were reviewed. The buffering techniques discussed in this section demonstrate efficiency in terms of time and memory. However, none of them guarantees optimality of the solution. In fact, the complexity of buffering problems generally makes it impractical to obtain an optimum solution in a reasonable time. In chapters 3 and 4, it will be shown that the optimum solution can be efficiently found for a particular class of buffering problems, known as *balanced buffering*. The efficiency of the new algorithm proposed in this thesis has been proved through runtime tests. Moreover, the elegant problem formulation in this thesis provides a good foundation for future work where the method presented can be used in solving more complex buffering problems.

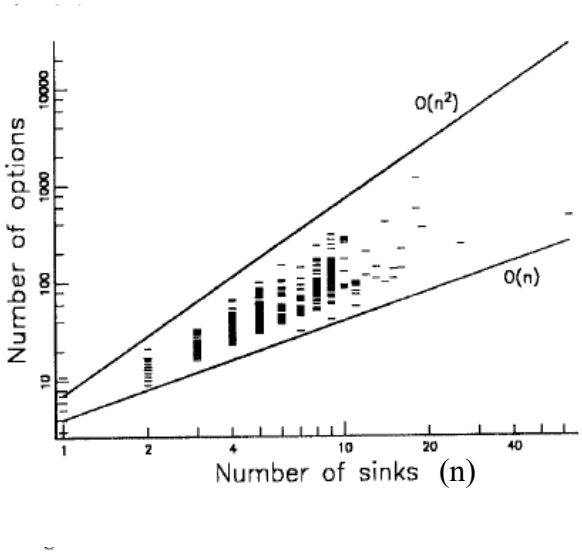
Method	**Time Complexity**
Van Ginneken's algorithm 	$O(n^2)$
Extension of Van Ginneken's algorithm for multiple buffer insertion [HRKIC and LILLIS, 2002]	$O(b^2 n^2)$
[HRKIC and LILLIS, 2002]'s method improvement by [SHI and ZI, 2005]	$O(bn^2)$
Van Ginneken's algorithm speedup by [SHI and ZI, 2005]	$O(n \log^2 n)$
Van Ginneken's algorithm speedup with multiple buffer insertion [SHI and ZI, 2005]	$O(b^2 n \log^2 n)$
Simultaneous routing buffer insertion under fixed buffer location [CONG and YUAN, 2000]	$O(n^3 \log n)$

Table 2-1 Comparison the time complexity of Van Ginneken's algorithm and its variations

3 BALANCED BUFFERING

In this chapter a specific buffering problem is addressed and a new solution is proposed. The objective of the buffering method introduced is to generate the fastest buffer tree for a set of identical fanouts,(identical in terms of capacitance and required time). Due to the symmetrical buffer tree structure obtained by the proposed method, it is called *balanced buffer tree* while the process of producing a balanced buffer tree is known as *balanced buffering*. Analyzing this type of buffering from a mathematical point of view and extracting elegant formulas expressing the characteristics of the best buffer tree, a new buffering algorithm is proposed in this chapter[1]. The presented algorithm ensures solution optimality due to the nature of the applied method which is branch-and-bound. While the tests provided in this chapter show a reasonable time and space complexity for the presented balanced buffering algorithm, a number of techniques will be introduced in chapter 4 to improve the performance of this algorithm. This chapter also encompasses some exclusively tailored data structure to the proposed problem-solving method. To strengthen the purpose of using an idea or a technique, adequate mathematical proofs are provided in appendices.

Chapter Outline

In section 3.1, the balanced buffering problem is discussed and its major characteristics are studied. Section 3.2 gives a formal definition of the balanced buffering problem. Section 3.3 examines the required conditions of using the branch-and-bound method to design the buffering algorithm and the structure of the feasible region (search space) explored by the buffering algorithm is examined. Section 3.4 presents the flowchart of the balanced buffering algorithm. In section 3.5, it is explained how solutions are handled by the algorithm. Section 3.6 studies the topology of balanced buffer trees and the proper ways of making balanced sub-trees to implement the branch-and-bound algorithm. In section 3.7 two rules are introduced to help avoiding non-promising solutions. Section 3.8 presents efficient techniques utilized to produce a neutral-phase buffer tree with a buffer library containing both inverting and non-inverting buffers. Finally, section 3.9 shows the runtime of the proposed buffering algorithm.

[1] The basic mathematical and algorithmic techniques discussed in this chapter have been investigated by [AMOURA and MAILHOT, w. d.].

3.1 Balanced Buffering Applications: Facts and Potentials

As mentioned before, buffering is used at both logical and physical design levels to improve circuit timing and help decoupling large fanouts. In a typical buffering problem, the load specifications are not necessarily identical, i.e. the *required times* in logic synthesis and the realistic *load capacitances* obtained in physical design are regularly very diverse. However, certain types of circuitry do exhibit local specifications where loads have similar input capacitance and required time. This is the case where a balanced buffer tree is generated. In figure 3.1 two types of buffering problems are compared. Each box represents a load and the size of the box represents the load capacitance. In addition, the distance between the boxes and the source gate is used to depict their required times. The closer a box is to the source gate, the smaller its required time.

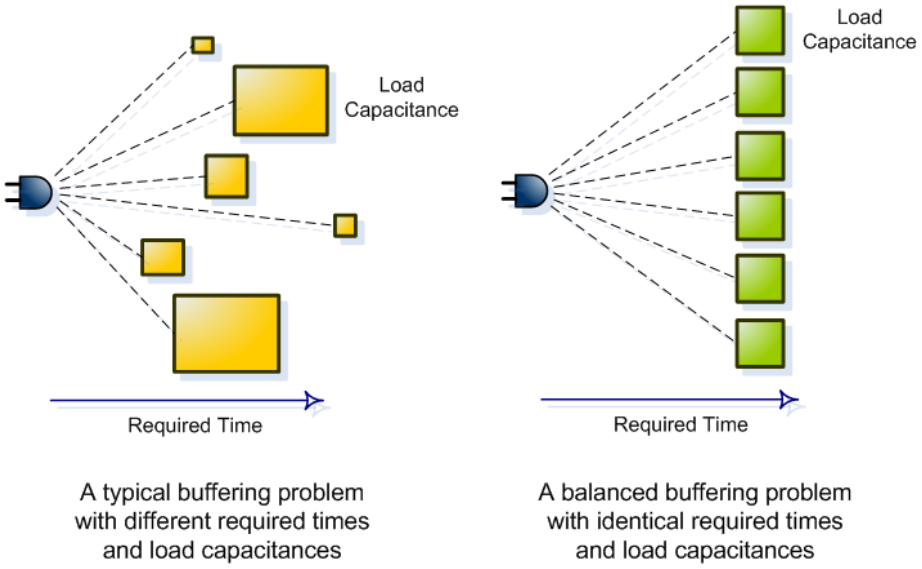
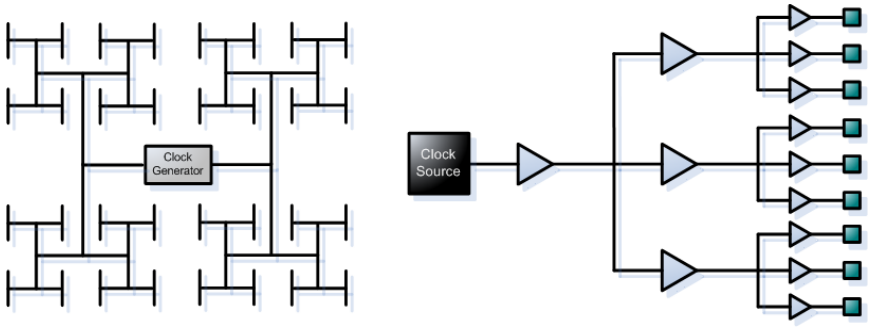


Figure 3-1 Balanced buffering versus a typical buffering problem.

A good example of balanced buffer trees is in clock tree construction. The purpose of a clock signal in a synchronous digital design is to define a reference for data movement. Hence, the stability of clock signals is extremely important. One solution to distribute signals with minimal skews and healthy signal waveforms is to generate a tree with an H-form structure where the distances from the center to all branch points are the same, and hence, the signal delays would be the same. Typically, such a clock network should be a balanced buffer tree. Note that due to routing constraints and different fanout requirements, this is difficult to implement in practice, but certain methods have some preferences on

timing objectives rather than routing targets, and therefore, they use H-form clock trees as a pre-routing operation. H-form structure and a balanced buffer clock tree are shown in figure 3-2.



a) H-form structure for a clock tree b) Balanced buffer clock tree

Figure 3-2 Clock tree construction

Yet, this is not the only application for which this thesis proposes a buffering method. The long term goal of this work is for a designer to be able to solve even unbalanced buffering problems for more general cases by casting them into the balanced buffering model. However, in this thesis only balanced buffering problem is solved, the generalized techniques being left for future work.

Modeling of the problem is an important step toward solving it. If available, a mathematical model which can capture the basic characteristics of the problem is preferred because it can significantly simplify both developing and implementing the solution. In order to extract a mathematical model for the balance buffering problem it is therefore useful to find the minimum delay achievable for a typical buffering using a balanced buffer tree. More precisely, answering the following question leads to an elegant mathematical model:

Given a source and a set of identical loads with similar required time and capacitance, what is the best delay achievable by an ideal buffer tree?

The term *ideal buffer tree* means a buffer tree which exhibits prefect timing properties but is not necessarily implementable. While in a typical tree branching factors and the depth of the tree are expressed by integer values, in an ideal buffer tree these parameters may be non-integer and hence make it impossible to implement such a tree. In this dissertation, the term *ideal buffer tree* will be used as opposed to *real buffer tree* which is an implementable tree. Also, *Ideal delay* and *real delay* will be used throughout this dissertation to indicate the delay values for each of the mentioned buffer tree

types. For more clarity, a statement of the problem follows, to determine the overall constraints and objectives with which the problem is going to be solved.

3.2 Statement of the Problem

The goal is to maximize the slack of the circuit, or simply minimize its delay. The physical properties of the source gate and its fanouts are given, and all fanouts are identical in terms of required times and physical characteristics. No routing topology is given and it is possible to freely look for the best buffer tree topology meeting the timing needs. It is preferred to use a buffer library consisting of inverting and non-inverting buffers to help finding better solutions to the problem. No buffer sizing is applied to reduce the circuit delay. The generated buffer tree must have a positive phase, i.e. in the presence of inverters no phase shifting is allowed to occur and the buffer tree must have no effect on the circuit logic. Thus, the problem is simply stated as:

$$\textbf{\textit{Minimize } } D_N$$

Where D_N denotes the overall delay of a network consisting of a source and its fanouts.

3.3 Method

Since the significant work of Van Ginneken [VAN GINNEKEN, 1990] which proposed a dynamic programming algorithm for inserting buffers and the work of Touati [TOUATI, 1990], many of the practical buffer insertion techniques in use today have been proposed to extend these algorithms or improve their time complexities. However, even though general buffering has been proven to be an NP-complete problem [BERMAN et al, 1989] [SHI et al, 2004], exhaustive search techniques, such as branch-and-bound algorithm, are more effective than dynamic programming in solving balanced buffering problems, due to the nature of such problems which exhibits good branching and bounding properties. In fact, searching for the best topology as well as the best buffer arrangement makes balanced buffering extremely difficult to solve with a dynamic programming technique. This is due to the fact that the efficient structure required to cast a balanced buffering problem into a dynamic programming pattern is hard to find as the best buffering configuration varies with topology changes. Therefore, a branch-and-bound method is chosen to cover all possible solutions to a balanced buffering problem. The branch-and-bound method was first introduced in 1960's in the Operations Research community [LAND and DOIG 1960]. It is discussed extensively in [MICHALEWICZ and FOGEL 2000] [AHO et al, 1983] and only the

implementation issues are discussed in this dissertation. Two basic tools are required for any branch-and-bound procedure to be implemented:

1) A **branching** technique [1] to divide the feasible region[2] of possible solutions into feasible sub-regions.
2) A **bounding** tool to quickly find the lower bound (in minimization problems) or upper bound (in maximization problems) within a feasible region.

It is understood that here the problem is a minimization problem, as a buffer tree with the minimum delay is going to be constructed. To find the solution in a minimization problem, the main approach is to observe when the *lower* bound for a sub-region **A** from the search space is greater than the *upper* bound for any other previously examined sub-region **B**. In that case, **A** is safely discarded from the search. This is called **pruning** in the branch-and-bound method. To implement pruning, it is usually necessary to find a global variable keeping the minimum upper bound seen among all sub-regions examined so far. Using this global variable, any solution that has greater lower bound can be discarded. The minimum delay achieved so far can be used as the required global variable in balanced buffering.

In a minimization problem *cost* is addressed, as opposed to maximization problems where function *value* is addressed. The cost of each buffering solution in the feasible region is the delay of the buffer tree. The search procedure in branch-and-bound is usually terminated when every possible solution in the feasible region is either pruned or examined. To have the final answer, a record, or *snapshot*, of the best buffer tree is always kept. A snapshot is taken whenever a solution with a smaller delay than the best previous solution is encountered. The runtime efficiency of the branch-and-bound algorithm is highly dependant on branching and bounding methods. Bad choices can lead to repeated branching without pruning or sub-optimal feasible region where optimum solutions have not been generated. An efficient branching method and a bound equation are going to be introduced in section 3.3.1 and 3.3.2.

3.3.1 Recursive structure

In order to efficiently split the feasible region into feasible sub-regions, it is necessary to define a recursive structure for the balanced buffering problem. The following definition yields such a structure:

[1] Note that *branching technique* is used in the branch-and-bound algorithm and it must not be confused with *branching factor* by which a tree is divided into sub-trees.
[2] In branch-and-bound, feasible region is a term referring to a set of admissible values for a given function.

Given a source gate and a set of sinks, the fastest buffer tree is the combination of fastest balanced buffered sub-trees with a certain branching factor, where the root of every sub-tree is a buffer taken from a buffer library.

Figure 3-3 shows how a set of sub-trees are generated by a branching factor of 3 to the original tree. Branching factor is always equal to the number of sub-trees. While it can be a floating point value for ideal buffer trees, it must be an integer value for a real buffer tree.

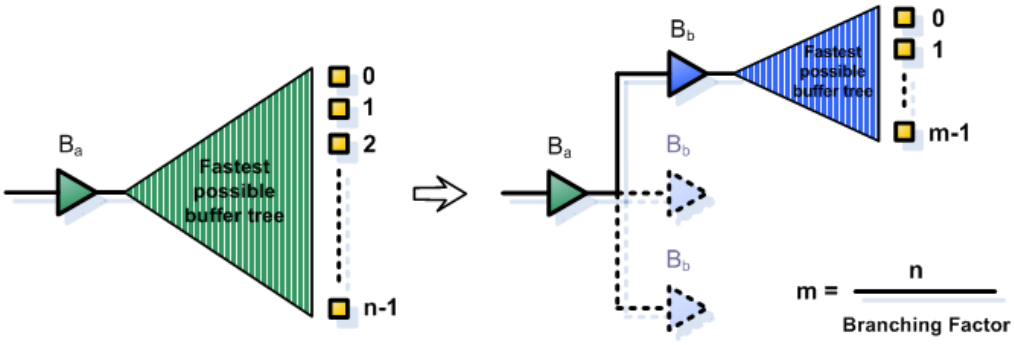


Figure 3-3 Recursively built sub-problems

3.3.2 Lower bound equation

In order to implement the branch-and-bound algorithm, an effective lower bound is necessary. It is sufficient for the underlying delay model used for the bound calculation to be the simplistic Elmore delay formula [ELMORE, 1948], where no wire delay is taken into account. The Elmore delay model basically overestimates the delay of a tree, the effect of which is on the algorithm runtime and not on the solution quality. In order to use the Elmore delay model, the following definitions is used:

Definition. *Gate delay:* the delay at gate ***g*** from input pin ***i*** is defined to be the delay from that input pin to the input pins of its fanout.

Definition. *Elmore delay:* For a gate driving a capacitive load, the Elmore delay is calculated using the following equation:

$$Delay = \alpha + \gamma \beta$$

Where α is the intrinsic delay, β is the output resistance of the driving gate, and γ is the input load capacitance of the load. Using the Elmore delay formula, figure 3-4 shows an example of calculating the gate delay for a given buffer tree with 1 driving and 2 driven buffers.

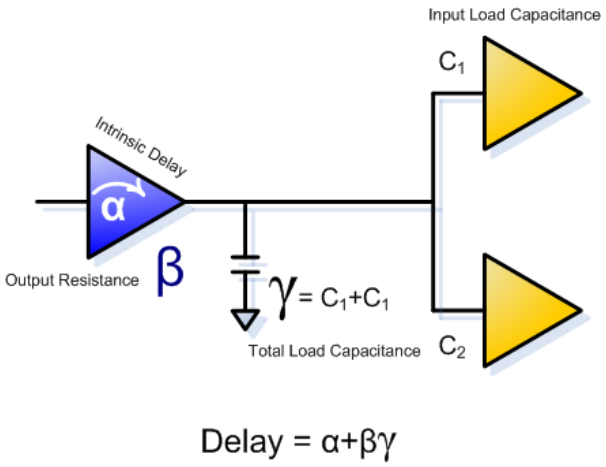


Figure 3-4 Calculating Elmore delay for a given buffer tree

Definition. *Best buffer:* the buffer that forms the ideal buffer tree.

Having defined the required concepts, it is now possible to introduce the ideal delay. One can calculate the ideal delay of a buffering solution before solving it using the information about the driving buffer and load capacitance using equation 3-1:

$$D_{optimum} = \mu\left(1+\ln\left(\frac{\beta_0\gamma_T}{\mu}\right)\right) \qquad 3\text{-}1$$

Where $D_{optimum}$ is the ideal delay of the ideal buffer tree, β_0 is the output resistance of the source gate, γ_T is the total load capacitance and μ is defined in equation 3-2:

$$\mu = \beta_b\gamma_b e^{\left(\frac{\alpha_b}{\mu}+1\right)} \qquad 3\text{-}2$$

Where β_b and γ_b are respectively the output resistance and the input capacitance of the best buffer in the library and α_b represents its intrinsic delay. See appendix A for more details on how the ideal

delay is calculated. See also appendix B for a proof that a unique best buffer exists. The best buffer is the one with the smallest μ in a given buffer library. This is proved in appendix C. One can systematically find this best buffer before starting to solve the problem and use its value when calculating the bound. Figure 3-5 shows the special structure of an ideal buffer tree. In that figure an ideal buffer tree with continuous (as opposed to discreet) branching factors and a buffer grid providing possible buffer arrangements are illustrated. While the ideal buffer tree is shown as a triangle, the real buffer tree for the same case has to be constructed only by using the paths provided by the buffer grid.

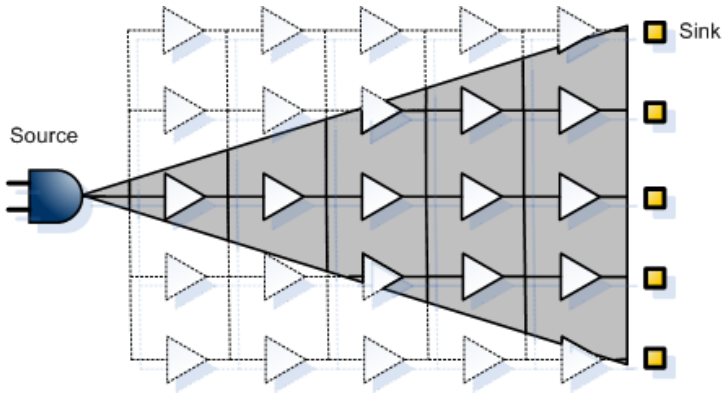


Figure 3-5 A non-discrete structure of an ideal buffer tree

The lower bound (minimum cost) of a buffering solution in the feasible region is calculated as follows in equation 3-3:

$$LowerBound = D_{current} + \alpha_b + Ideal\ Delay_{Subtree} \qquad 3\text{-}3$$

$D_{current}$ is the delay of the buffer tree up to the root of the sub-tree for which balanced buffering is going to be done, α_b is the intrinsic delay of the buffer at the root of the sub-tree and $Ideal\ Delay_{Subtree}$ is the ideal delay of the sub-tree obtained by equation 3-1. This is shown in figure 3-6.

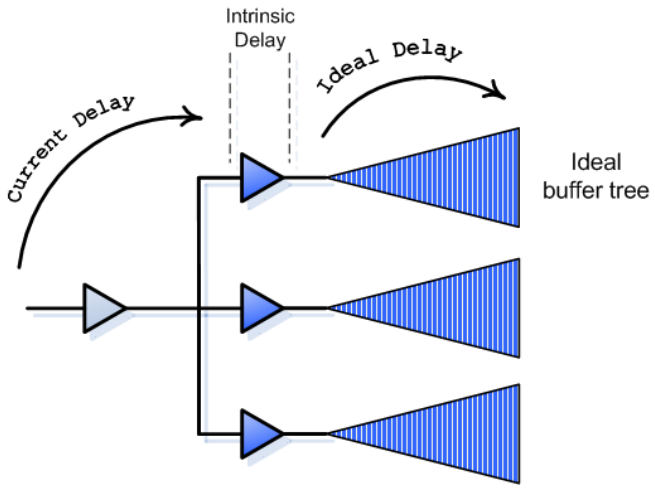


Figure 3-6 Lower bound calculation

The ideal delay is safe to use in lower bound calculations, since equation 3-1 assumes ideal conditions that cannot occur in real problems.

3.3.3 Search space

There always exists at least one best buffer tree for a given source and a set of loads. In a balanced buffer tree, all paths from source to sinks are equivalent. It is therefore possible to build the best tree hierarchically by finding and combining the fastest buffered sub-trees, provided that in the process all branching factors are explored. If a branching method is well defined, the feasible region ideally becomes a tree, and so the search space is called *search tree*. However, as will be discussed in chapter 4, the existence of many common sub-problems in solving a balanced buffering problem makes it less profitable to use a pure search tree structure. The real structure of the feasible region is in fact a directed acyclic graph where each node represents a solution. This is shown in figure 3-7 where common sub-problems are depicted darker than the others. Common sub-problems will be addressed in chapter 4.

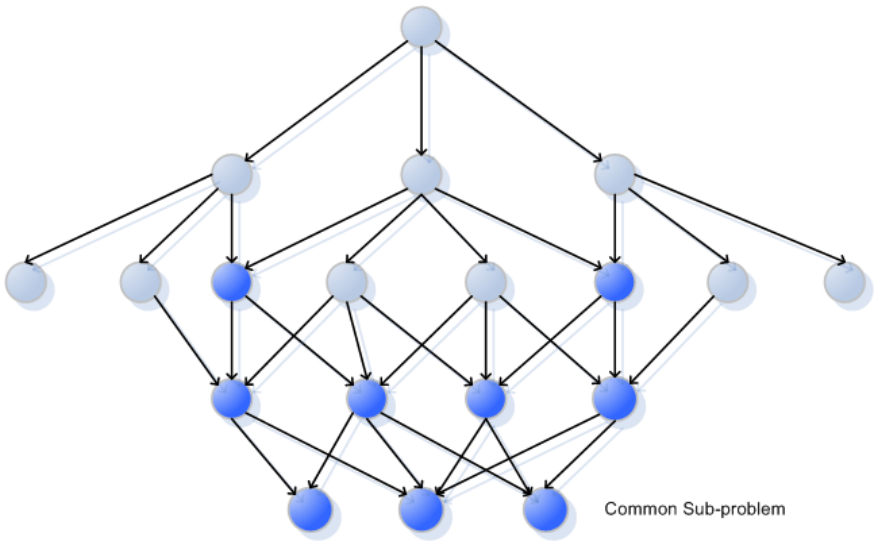


Figure 3-7 Feasible region as a directed acyclic graph.

Searching the best buffer tree in the feasible region corresponds to traversing the search space graph. While traversal progresses, non-bounded solutions are reached through some search path. A search path is in fact a scenario presenting a certain sequence of tree partitioning and buffer insertion applied to the original problem. Each node in the feasible region is a local buffering solution consisting of a branching factor and a buffer at the root of new sub-trees. In the end, the best scenario is the one that generates the fastest buffer tree. An example of a feasible region and the best scenario are given in figure 3-8. The search process starts with the root of the feasible region graph where no buffer is inserted. At each step, new buffering problems (new nodes) are produced by applying a local buffering solution to the previous buffering problem (previous node). Consequently, as the search traversal goes deeper in the feasible region graph, more buffers are inserted in the buffer tree and sub-trees become smaller. When a local buffering solution is generated, its lower bound is computed and compared to the global best delay. If this lower bound is greater than the best global delay, it will not be profitable to solve the new problem. This is the point where a portion of the feasible region is cut out by the bound, as shown by dashed lines in figure 3-8. In addition to the lower bound calculation, the real delay of the current buffer tree is calculated and compared to the best global delay every time that a node is generated. An update of the best global delay is needed when the current buffer tree delay is better than the current best global delay. The best scenario is a path from the root of the search graph to the node that has the minimum real delay.

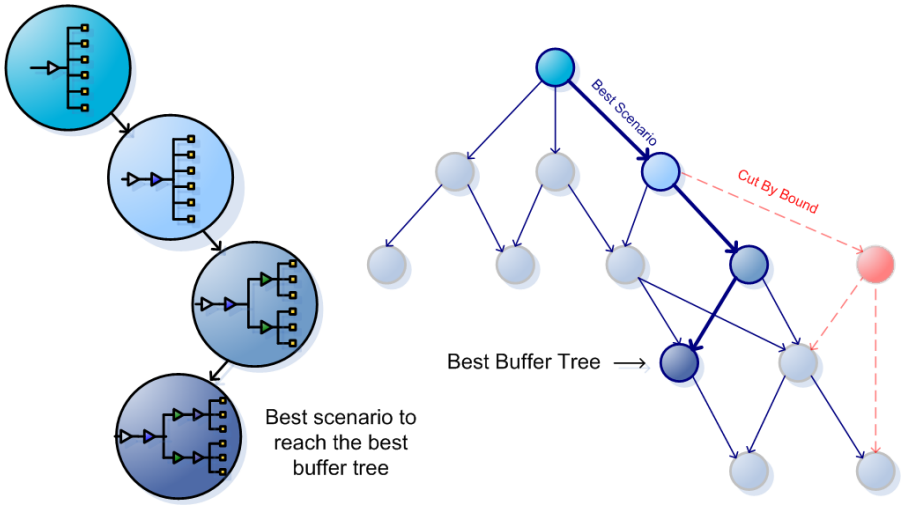


Figure 3-8 Search space graph

3.4 Algorithm

In this section the balanced buffering algorithm is presented based on the branch-and-bound method and using the materials explained in section 3.3. This algorithm enumerates every possible solution to a balanced buffering problem, calculating only those solutions allowed by the bound. Thus, one expects to achieve a high quality solution, i.e. the fastest buffered tree together with the best tree topology. Given a source output resistance and a set of identical sinks, the algorithm calculates the minimum delay of the best ideal buffer tree and uses this value to prune non-promising solutions in the search space. The exhaustive nature of the search method guarantees its optimality, and appropriate use of the bound significantly reduces the number of potential solutions that really need to be visited. The flowchart of the balanced buffering algorithm is given in figure 3-9. Due to the recursive structure identified for the branching procedure, it is best to design a recursive algorithm. It hierarchically produces sub-problems and either solves them or prunes them. The solutions to a buffering problem are stored in a solution list and explored one by one. The *stop* condition of the recursive algorithm is true either if the current problem is pruned by the bound or the end of a solution list is encountered. During this procedure, the best solution is detected and saved whenever it yields a smaller delay than the best delay found so far. The real delay of the current buffer tree is calculated immediately after the beginning of the recursive function. Therefore, it is possible to calculate and use the lower bound to avoid solving a non-promising problem before any solution list enumeration. At each search step, the current buffer tree can be constructed by orderly combining the solutions kept in a stack. This solution stack is then used in saving the best buffer tree by saving a copy of it, and is used at the end of the search to generate the final best tree.

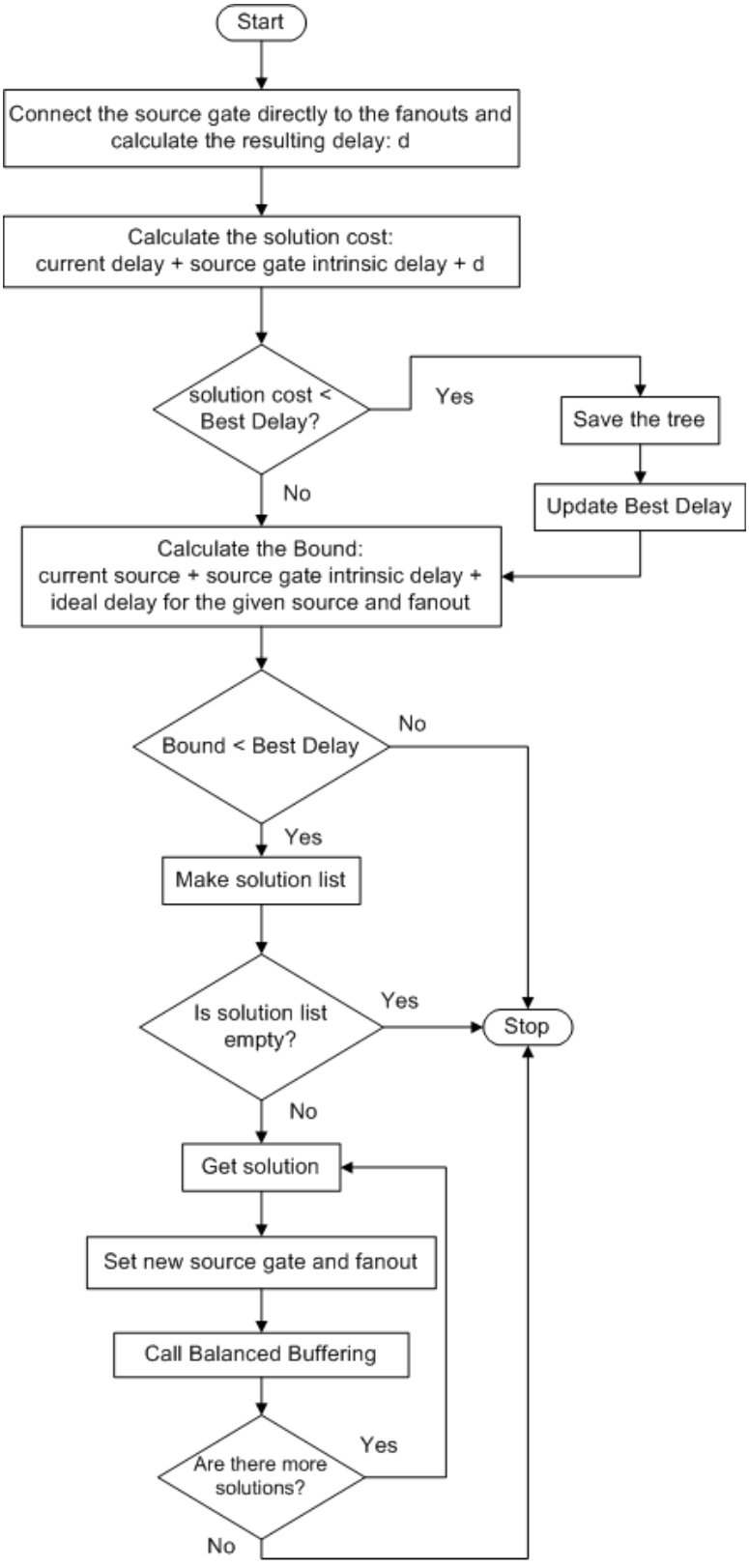


Figure 3-9 Flowchart of balanced buffering algorithm

An example of running that function is illustrated in figure 3-10. In this example the original problem has 12 fanouts. The best buffer tree obtained for that problem contains 4 local solutions kept ordered on the stack. When, for example, solution 1 is applied to the original problem, 3 new sub-trees with 4 fanouts are formed and buffer *B1* is used at the root of sub-trees. Each of those 3 new sub-trees form a new buffering problem and is solved by applying solution 2. The rest of the solutions are used in a similar manner to build up the buffer tree.

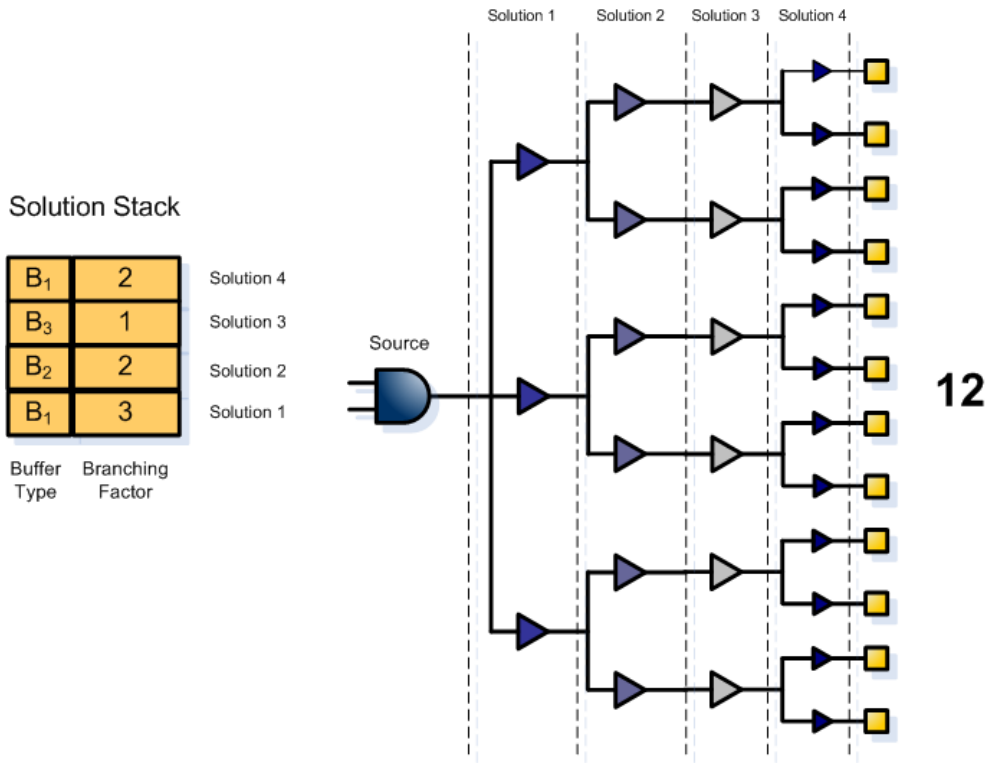


Figure 3-10 Saving and reconstructing the best buffer tree

A set of data structures designed to efficiently run the algorithm is now introduced. The solution list enumeration will be shown, together with its use in exploring the search space.

3.5 Solution List

The set of solutions for a given source gate and a given fanout is a group of sub-trees that can be extracted from the original tree and solved as sub-problems. Each sub-tree has a source buffer at its root which is taken from the buffer library, and has a carefully chosen number of fanouts that gives it a

balanced form. In general, solutions are obtained by systemically producing possible sub-trees and inserting a buffer at their root. These buffers are taken from the buffer library. To have an efficient traversal ordering for the solution list, the ideal delay of each (buffer, sub-tree) pair is calculated and saved together with the source buffer and the branching factor to form a solution. The ideal delay can then be used as a measure to solve the promising solutions first. To that effect, the solution list is sorted in terms of ideal delays after all solutions are enumerated. This makes it possible to find more promising solutions with smaller ideal delays, placing them in higher ranks in the list. As a result, the solutions are explored in a top-down fashion, solving first the potentially better sub-problems. A generic solution and a solution list are illustrated in figure 3-11.

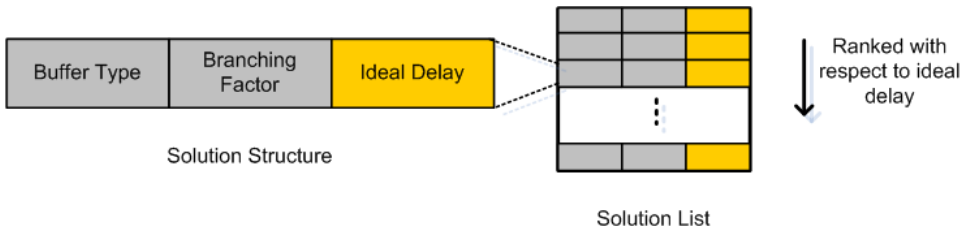


Figure 3-11 A generic solution structure.

3.6 Balanced Sub-trees

A set of balanced sub-trees is composed of those that have zero or at most one fanout in difference. The former case is called *strictly* balanced sub-trees whereas the latter case is called *partially* balanced sub-trees. In other words, strictly balanced sub-trees are obtained whenever the proper divisors of the fanout number is used to divide the tree, whereas partially balanced sub-trees are obtained whenever the fanout number is divided by a number which is not its proper divisor. The objective of defining a balanced structure is first to preserve the generality of the problem-solving method and second to be able to apply the solution to a sub-problem for other sub-problems at the same level. This saves significant runtime as only a portion of a buffering problem is being solved at a time, and the solution to the remaining sub-trees can be found by mirroring what is achieved for the sub-problem being solved. In figure 3-12 3 different ways of making sub-trees are shown.

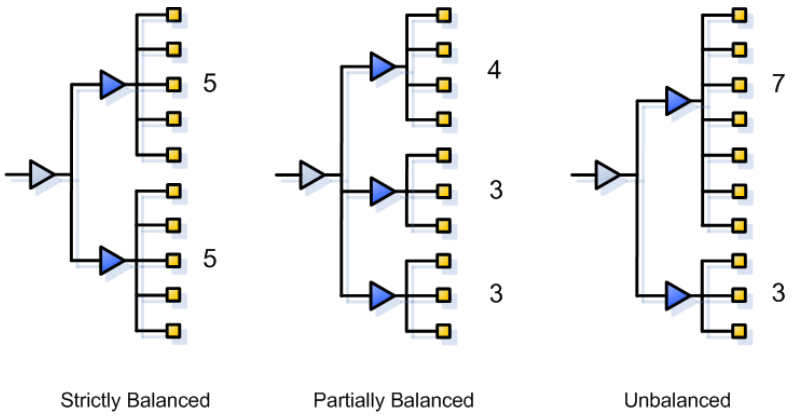


Figure 3-12 Splitting a tree with a fanout number of 10 in 3 different ways

Splitting a tree into balanced sub-trees is a very critical task. Bad choices can delay finding the best buffer tree in general and in particular make it impractical to use the same solution to all sub-trees at the same level. To split a tree, both strictly and partially balanced sub-trees are considered. However, in order to minimize the resource loss caused by mirroring a solution for dissimilar sub-problems, only one specific type of partially balanced sub-trees being generated is permitted. The necessary property of such group of sub-trees is defined as follows:

*If **n** represents the total number of partially balanced sub-trees, there must exist **n-1** instances of the sub-tree with the larger fanout number and only **1** instance of the sub-tree with a smaller fanout number.*

In figure 3-13 this specific case is shown. If there are n sub-trees, every sub-tree must have m fanouts except for the last one which has m-1 fanouts.

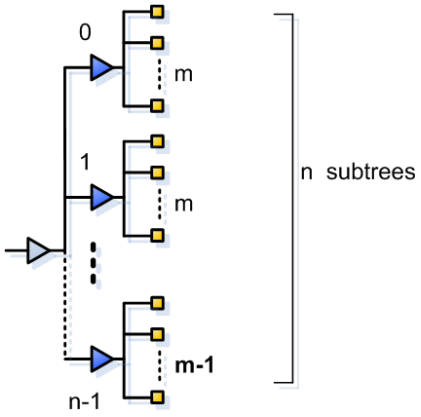


Figure 3-13 The specific case of partially balanced sub-trees

The splitting method introduced presents a group of sub-trees where all but one sub-tree have identical fanout numbers. This type of tree partitioning minimizes the number of enumerated sub-trees in producing partially balanced sub-trees, the effect of which is to reduce resource loss. It also helps achieving more realistic solutions by giving smaller critical path delays. The delay value of the critical path is directly influenced by the number of sub-trees. It can be shown that using strictly balanced and the specific case of partially balanced sub-trees minimizes critical path delay. This is due to the fact that the two classes of balanced sub-trees mentioned have maximum regrouping factor, i.e. minimum number of sub-trees. As shown in figure 3-14, the larger the number of sub-trees, the larger the number of buffers, and as a result, the larger the capacitance seen at the root of the tree. For example, one can split a tree with 17 fanouts by either 5 sub-trees with 3 fanouts together with a sub-tree with 2 fanouts (left buffer tree in figure 3-14), or 3 sub-trees with 3 fanouts and 4 sub-trees with 2 fanouts (middle buffer tree in figure 3-14). In the former case there are a total of 6 sub-trees and hence the total number of buffers used is 6, whereas in the latter case, there exist a total of 7 sub-trees and hence the total number of buffers used is 7. It is then trivial to observe that the capacitance seen at the root of the buffer tree is larger in the second case as also shown in figure 3-14.

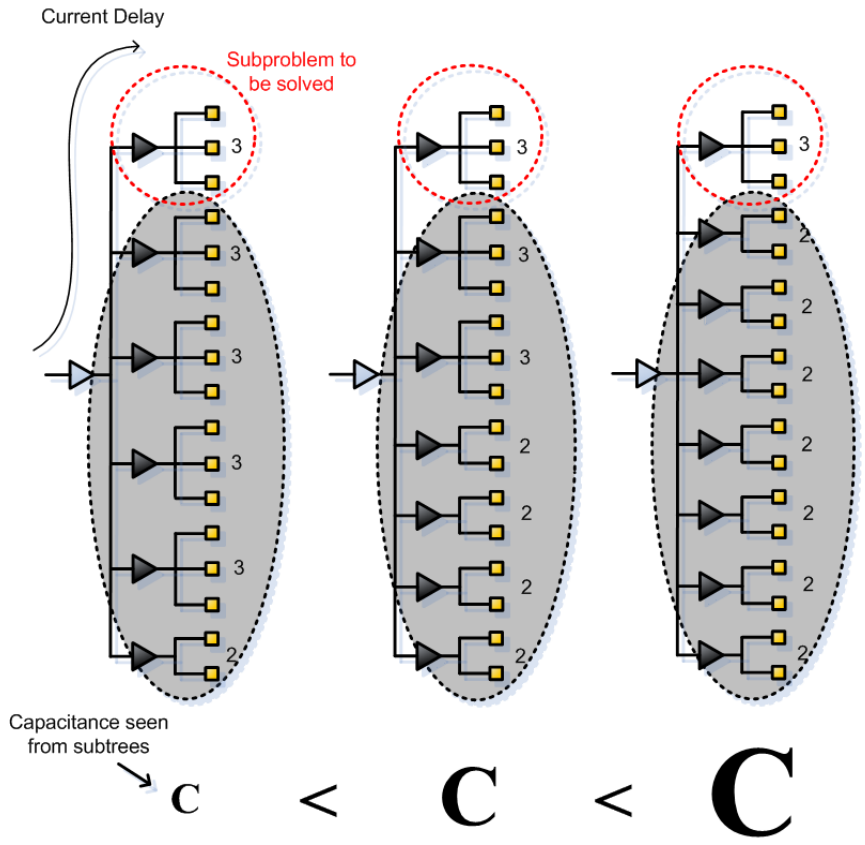


Figure 3-14 3 ways of making partially balanced sub-trees for 17 fanouts

As critical path passes through the root of the buffer tree, larger sub-tree capacitances seen at the buffer tree root lead to larger critical delay values. Therefore, one may obtain different lower bound values for the same sub-problem chosen to be solved as this critical path delay is used in the lower bound calculation as *current delay*. Choosing the right branching factor that yields minimum number of sub-trees is therefore a simple way to minimize the critical path delay and help finding more realistic lower bound values.

Given a fanout number N, the following method is then suggested to find the best branching factors for the balanced sub-trees:

Step 1) Find divisors of N and call them div $_N$. Consider all divisors in div $_N$ as a branching factor.

Step 2) Find divisors of N-1 and call them div $_{N-1}$.

Step 3) Find divisors of N+1 and call them div $_{N+1}$.

Step 4) For each divisor belonging to div $_{N-1}$, called d $_{N-1}$, if the quotient of N/ d $_{N-1}$ has a fractional value greater than or equal to 0.5, consider d $_{N-1}$ as a branching factor.

Step 5) For each divisor belonging to div $_{N+1}$, called d $_{N+1}$, if the quotient of N/ d $_{N+1}$ has a fractional value greater than or equal to 0.5, consider d $_{N+1}$ as a branching factor.

Let us find the balanced sub-trees of a tree with a fanout number of 15. First, divisors of 15 are listed, as well as the divisors of its adjacent numbers, 14 and 16. In order to have a set of strictly balanced sub-trees it is sufficient to simply use the proper divisors of 15, which are 1,3,5 and 15. Partially balanced sub-trees are also produced using the divisors found during steps 2 through 5, which are 2 and 8. A table of all strictly and partially balanced sub-trees is set up and shown in figure 3-15.

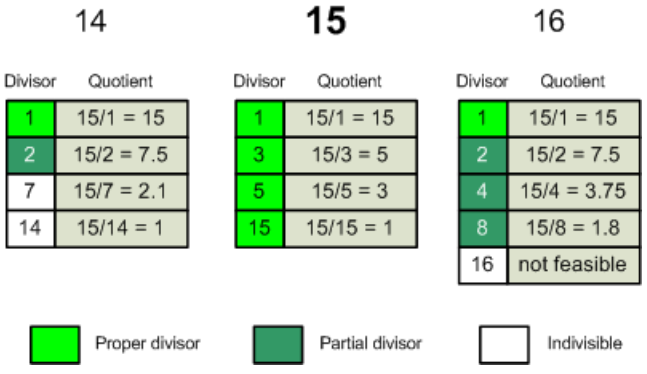


Figure 3-15 Legal divisors of 15.

A different problem representation is illustrated figure 3-16 for more clarification. In this figure the numbers inside the triangle represent the fanout number of the sub-trees, whereas the numbers in the left column outside the triangle represent the applied divisors. Solutions are those triangle rows that correspond to strictly and partially balanced sub-trees. An arbitrary row (which would not be chosen as a solution because it has 3 sub-trees which differ from the others) is shown with the corresponding buffer tree which it would entail.

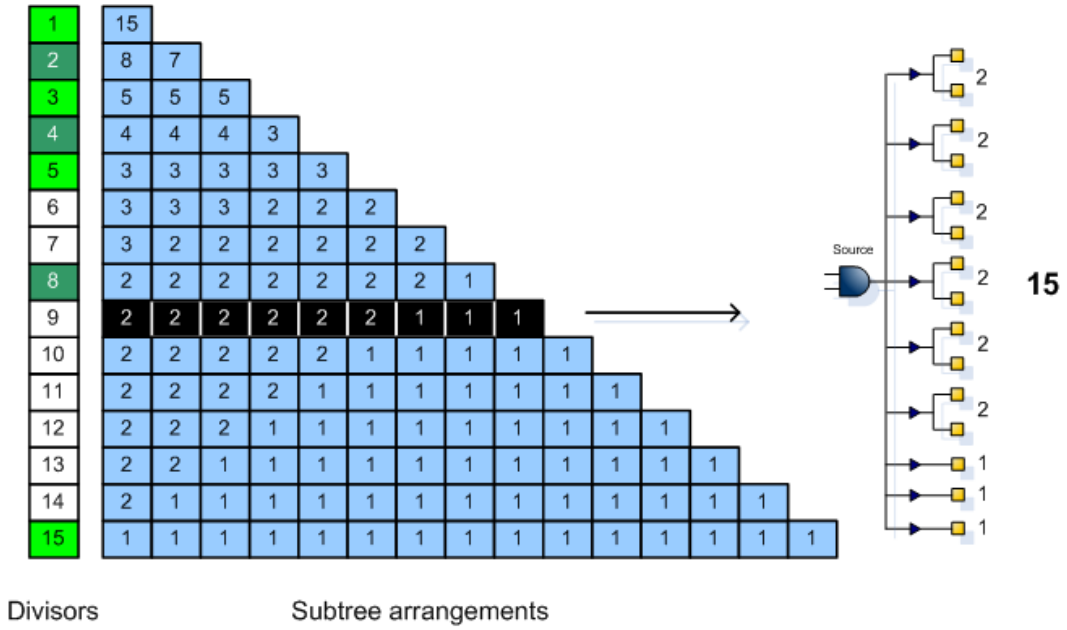


Figure 3-16 All possible sub-trees for a fanout of 15.

3.7 Buffer Selection

Certain enumerated solutions can be discarded under certain conditions even before exploring them in the feasible region. This filtering can occur at the time of creation of a solution list. The decision to keep or discard a solution can be taken for sequences of single buffers. It is often the case that a sub-tree has a fanout number of 1 (no splitting is actually happening) where just another buffer is added to the root of the tree. The following theorems state under which conditions solutions are definitely dominated by some others.

Theorem 1. Consider the case of having two different buffers (B2 and B3) placed along the same wire in figure 3-17:

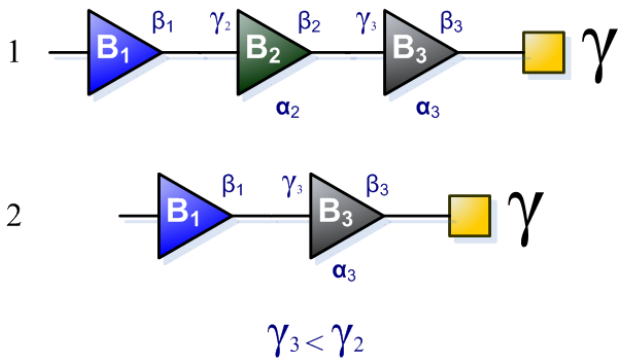


Figure 3-17 Single wire buffering illustrating theorem 1

Regardless of the physical characteristics of B1, if the input capacitance of B_3 is smaller than or equal to the input capacitance of B_2, buffer B_2 can simply be removed from the wire in order to have a better delay. This choice is independent from any other physical characteristics of the two buffers, such as intrinsic delay, output resistance, size, etc.

Proof. The claim is easily proven by writing the delay equations for each of the cases. In the following equations the input capacitance of each buffer is represented by γ_i, while the output resistance and the intrinsic delay of such buffers are respectively shown as β_i and α_i. The load input capacitance is also represented by γ.

$$D_1 = \beta_1\gamma_2 + \alpha_2 + \beta_2\gamma_3 + \alpha_3 + \beta_3\gamma$$
$$D_2 = \beta_1\gamma_3 + \alpha_3 + \beta_3\gamma$$

If D_2 is subtracted from D_1:

$$D_1 - D_2 = \beta_1(\gamma_2 - \gamma_3) + \alpha_2 + \beta_2\gamma_3$$

If $\gamma_3 \le \gamma_2$ it is always true that $D_1 - D_2 > 0$, or $D_1 > D_2$. This means sequences of buffers must always be done such that their input capacitance is monotonically increasing.

Theorem 2. Consider the case of having two different buffers (B2 and B3) placed along the same wire in figure 3-18:

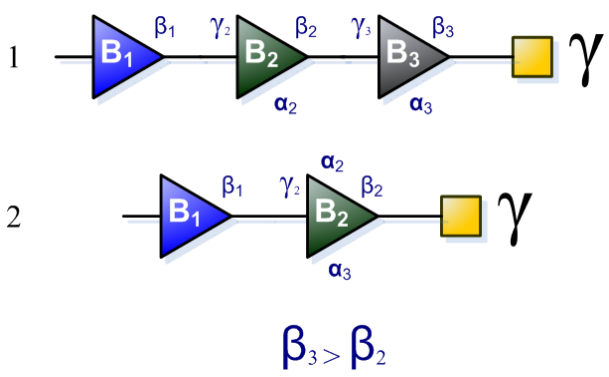


Figure 3-18 Single wire buffering illustrating theorem 2

Regardless of the physical characteristics of B1, if the output resistance of B_3 is greater than or equal to the output resistance of B_2, B_3 can simply be removed in order to get a better delay value for the whole circuit. This choice is independent from any other physical characteristics of the two buffers, such as intrinsic delay, input capacitance, size, etc.

Proof. The same notation is used as in theorem 1 to illustrate the physical characteristics. Then:

$$D_1 = \beta_1\gamma_2 + \alpha_2 + \beta_2\gamma_3 + \alpha_3 + \beta_3\gamma$$
$$D_2 = \beta_1\gamma_2 + \alpha_2 + \beta_2\gamma$$

If D_2 is subtracted from D_1:

$$D_1 - D_2 = \beta_2\gamma_3 + \alpha_3 + (\beta_3 - \beta_2)\gamma$$

If $\beta_2 \le \beta_3$ it is always true that $D_1 - D_2 > 0$, or $D_1 > D_2$. This means sequences of buffers must always be done such that their output resistance is monotonically decreasing.

Corollary 1. Theorem 1 and theorem 2 indicate that when buffers are inserted sequentially in a single line, the buffer input capacitances must appear in an ascending order while the buffer output resistances must appear in a descending order.

Corollary 2. Repeating the same buffer with exactly the same physical characteristics along a single wire never results in smaller delay. As a result, such solutions must never be included in the solution list.

3.8 Handling the Inverters

In order to improve the quality of the buffer tree generated, the buffer library is allowed to contain inverting buffers as well as non-inverting ones. Inverters, as discussed before, often present better timing and sizing characteristics than non-inverting buffers. However, the existence of the phase-shifting problem in the presence of inverters entails a more complicated algorithm design. It is now explained how inverters are taken into account such that no phase-shifting occurs in the final best buffer tree while the overall structure of the balanced buffering method still preserves its simplicity. To this end, a series of modifications are introduced in the structure of solution list, search space and the body of the main algorithm.

3.8.1 Changes in making a solution list

As inverters are now allowed to appear at different stages of a buffer tree, a sub-problem may make the entire phase of the tree become negative, by using an inverter at the root of a new sub-tree. On the other hand, a solution with a non-inverting buffer can also be the cause of a negative-phase buffer tree remaining negative, if an inverter has previously altered the phase somewhere before the current stage. This is shown in figure 3-19. Both of these solution types result in an impractical buffer tree with a negative phase, and hence, solving them must allow for correcting the phase problem.

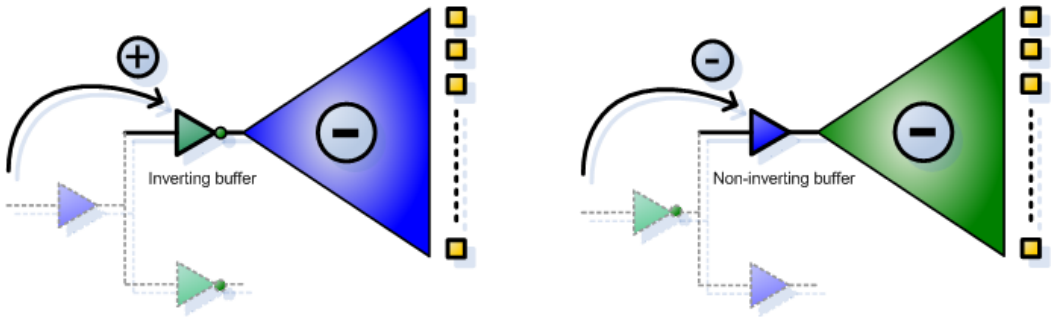


Figure 3-19 Two different sub-problems leading to a negative-phase sub-tree

These two solution types should not be discarded from the search space. Instead, a larger search space must be explored where solution lists are smartly re-organized for the negative-phase sub-problems. By setting up a proper sorting order for such solution lists, the traversal converges fast enough to make the runtime of the balanced buffering reasonably small. To that effect, for those sub-problems which produce or maintain a negative-phase, the solution list is broken into two sub-lists: one that resolves

the problem, suggesting the solutions which can change the phase back to positive, and the other sub-list containing solutions that have no effect on the buffer tree phase. The reader would probably guess what type of buffers each of these type sub-lists may contain: the former sub-list only includes inverters, and the latter sub-list consists of non-inverting buffers. It is important to note that the source buffer of a negative-phase sub-tree is not necessarily an inverting one. As said earlier, it could be non-inverting but still carry a negative-phase produced by preceding stages.

In solving a negative-phase sub-problem, the priority is on phase correction rather than finding the minimum delay, as opposed to a positive-phase sub-problem where the priority is merely on finding the minimum delay. As a result, the solutions of a negative-phase sub-problem are arranged such that the solutions whose phase can be fixed with inverting buffers appear first. To that effect, solutions with inverted phase are put together in one sub-list, and all solutions with non-inverting buffer are placed in a second sub-list. Calculating the ideal delay for every solution in both sub-lists, they are separately sorted in terms of minimum ideal delay. In the end, they are combined in a single list such that the sub-list with positive phase is placed on the upper half of the unified list and the other sub-list with non-inverting buffers is placed on the lower half. The combination of the two sub-lists is then saved as the ultimate solution list. An example of such *hybrid* list is shown in figure 3-20.

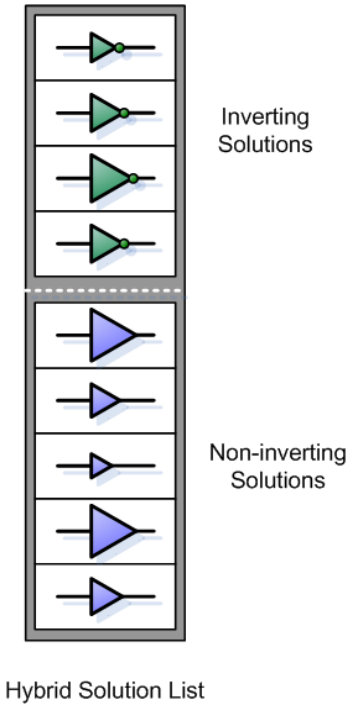


Figure 3-20 A hybrid solution list

When fetching solutions in the balanced buffering algorithm, those solutions that produce a positive-phase tree are taken first. Thus, the search procedure can find practical (positive-phase) buffer trees sooner, improving runtime. In addition, to allow fast convergence, a maximum of two identical inverters are allowed to be placed next to each other on a single wire, breaking the buffer selection rule introduced in corollary 2. This ensures an immediate solution available to a negative-phase sub-problem.

3.8.2 Changes in the search space structure

The existence of two classes of sub-problems, one with negative-phase and the other one with positive-phase, changes the search space structure. There are now two distinct regions in the search space, and every solution falls in either one of them. This is shown in figure 3-21 where the two connected circles correspond to the positive and negative phase search spaces, as also indicated by the plus and minus signs. In each search space there are a number of solutions represented by small circles with a buffer inside. A buffering solution and the new sub-problem made by this solution are connected together by an arrow.

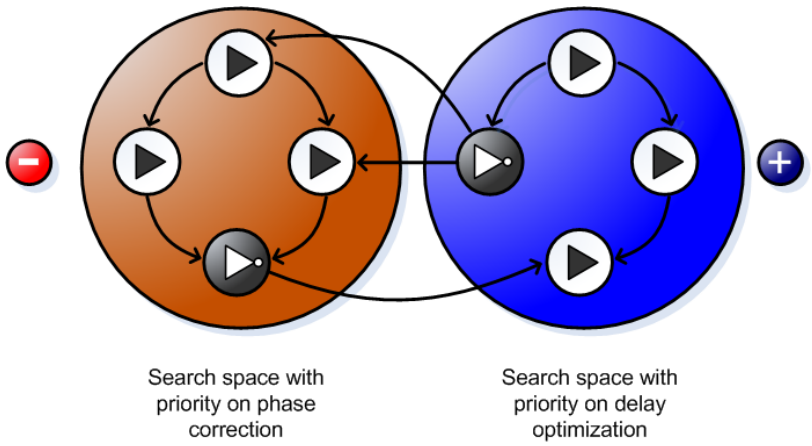


Figure 3-21 Two connected search spaces with different priorities

The graph traversal has now to be modified. The traversal must always begin and end in positive-phase search space. This is because the phase of the original problem is assumed to be positive and the constructed buffer tree must preserve the circuit logic. After beginning, the search traversal can either stay in the same search space using non-inverting buffers, or pass through the negative search space at some points using inverting buffers. Altering the search space then occurs only when a solution with

an inverting buffer is being solved. In contrast, as long as non-inverting buffers are used, traversal stays in the same search space, as no phase-change is encountered.

However, in the balanced buffering algorithm introduced before no extra consideration is required to handle the traversal of the double search space, except for saving the best buffer tree. Given that no buffer tree with a negative phase is of interest, the scenarios ending in the negative-phase search space are never saved.

3.8.3 Modified algorithm

In order to take phase shifting into account, an argument is added to the algorithm header indicating the current phase of the tree. This new argument must be calculated and assigned before every function call, regarding the phase of the current buffer tree and the phase of buffer type that has been selected from the solution list. In addition, the condition of having a positive-phase buffer tree candidate is verified before saving the best solution. In this manner, inverters are successfully taken into account using efficient solution lists and imposing the least modifications on the main algorithm.

3.9 Experimental Results

The proposed balanced buffering algorithm is implemented in C++. The program runs on a Pentium IV computer (CPU 3.4 GHz) and for different fanouts, ranging from 10 to 1000. Two buffer libraries are applied, one with 6 and the other with 7 buffers. These are libraries **A** and **B** in table 3-1 respectively. Both libraries contain inverting and non-inverting buffers. The intrinsic delay, output resistance and input capacitance for the buffers range from 3-21 ps, 0.03-0.15 Ω, 30-150 fF. The input capacitance of the load is set to 500 fF and the output resistance of the driving source is set to 0.5Ω. The runtime of solving those problems given the fanout numbers and buffer libraries is shown in table 3-2.

Buffer Library	Buffer ID	Type	Intrinsic Delay ps	Output Resistance Ω	Input Capacitance fF
A	1	Inverting	2	0.15	150
	2	Non-inverting	20	0.04	30
	3	Non-inverting	4	0.06	80
	4	Inverting	2	0.04	50
	5	Inverting	4	0.12	10
	6	Inverting	10	0.1	100
B	1	Non-inverting	7	0.09	110
	2	Inverting	2	0.04	50
	3	Inverting	12	0.03	60
	4	Non-inverting	8	0.05	150
	5	Inverting	2	0.05	40
	6	Non-inverting	2	0.1	130
	7	Inverting	15	0.04	40

Table 3-1 Buffer libraries

Fanout	Runtime (seconds)	
	Buffer Library A	Buffer Library B
10	8	3
20	45	16
30	107	54
40	153	86
50	188	135
60	444	366
70	477	224
80	628	421
90	1202	675
100	488	253
110	1136	641
120	2514	1424
130	960	909
140	2118	1264
150	1261	732
160	1764	1077
170	1769	1121
180	4863	3064
190	2926	1744
200	1936	1453
300	7665	5726
500	14466	23124
1000	59077	33491

Table 3-2 Runtime for different fanouts using different buffer libraries

3.9.1 Runtime Analysis

From tables 3-1 and 3-2 it is observed that program runtime generally increases with fanout number. However, one has to note that this increase in runtime is not monotonic. This is due to the fact that program runtime is directly dependent on the search space size rather than on the fanout number. If, for example, a tree with a fanout number of 90 is being buffered, it does not necessarily result in a smaller search space size than a tree with a fanout number of 100, as the set of divisors for each one have different sizes. It is therefore preferred to take the fanout divisors (branching factors), into account rather than the fanout number itself in studying the algorithm runtime.

Another reason for variation in the algorithm runtime is the choice of the search space traversal. With different search strategies the final solution may be found sooner or later. The effect of this is on the size and the time that the redundant portion of the search space is cut out by the lower bound. In the balanced buffering algorithm proposed in this thesis, a depth first traversal has been selected to explore the search space. However, finding more efficient search strategies is a suitable subject for future work.

In a branch-and-bound algorithm where the size of the feasible region is dependent upon the fanout number and its divisors, calculating the time complexity is not a straightforward matter. As a result, a regression analysis[1] is done in order to achieve estimation for the algorithm runtime. A linear regression is then performed with an objective of finding the least squares fitting value[2] for a set of fanouts ranging from 10 to 200 with an interval of 10, using buffer library **A**. The result of this operation is an estimated second-degree function of $y = 11.028 \text{ X } ^{1.907}$ with a least-square value of 0.94 which indicates a reliable estimation. The resulting trendline is shown in figure 3-22. One can therefore suggest an approximated time complexity $O(n^2)$ for the presented buffering algorithm.

[1] **Regression Analysis** estimates the relationship between variables so that a given variable can be predicted from one or more other variables [GAUSS 1809]

[2] The **Least-squared Value** is actually the square of the correlation coefficient. The correlation coefficient gives us a measure of the reliability of the linear relationship between two variables. Values close to 1 indicate excellent linear reliability.

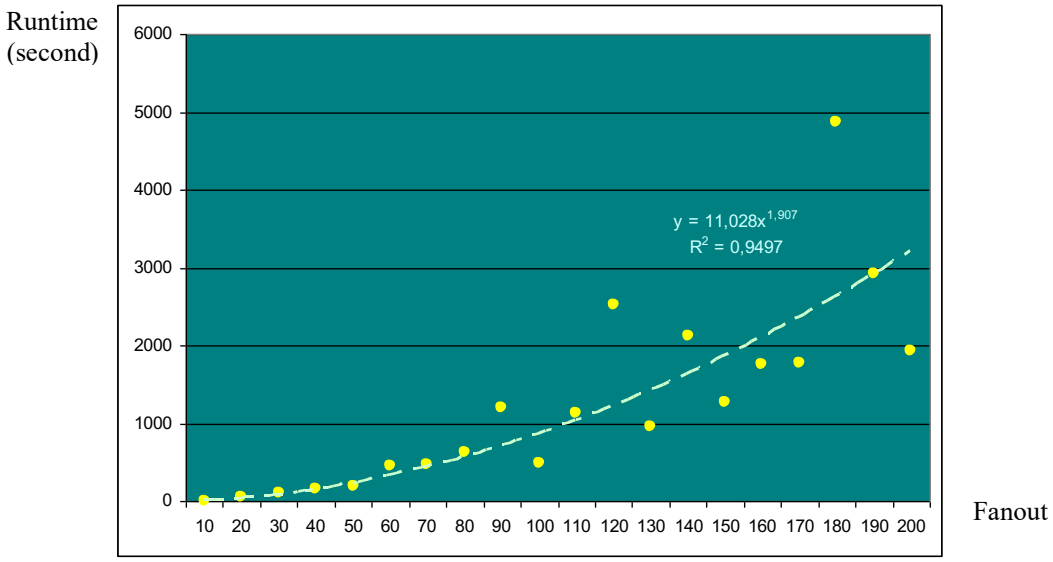


Figure 3-22 Curve-fitting on an arbitrary set of buffering problem runtime

3.10 Summary

A new buffering method was introduced to solve the balanced buffering problem where loads are identical in terms of required time and capacitance. A number of important buffering issues such as finding the best buffer tree topology, taking buffer libraries into account and handling the phase-shifting problem in the presence of inverting buffers have been addressed. Balanced buffering performance has not been compared to other buffering methods in use today. The reason for this is the nature of the branch-and-bound algorithm: the buffering method presented in this chapter guarantees solution optimality, while other approaches do not. Furthermore, the most advanced buffering techniques are part of proprietary systems where it is impossible to isolate a specific technique (here, buffering) from the other optimization operations that take place in those systems. As a result, a number of techniques will be put together in chapter 4 to ensure an efficient buffer insertion algorithm by improving its runtime and memory consumption while the solution optimality is also preserved. Although there are some applications for balanced buffering, the ultimate goal is to apply the basic ideas of balanced buffering to solve more general buffering problems in future. The underlying mathematical and algorithmic structure has been developed in this chapter for further extensions of the subject. A brief summary of the chapter contents is provided in table 3-3.

Problem : Balanced Buffering			**Solution: Searching for the Fastest Buffer Tree**			
Required Time	Load Capacitance	Topology	Delay Model	Method	Buffer Library	Inverters
Identical for all loads	*Identical for all loads*	*Not given*	*Elmore's*	*branch-and-bound*	*Considered in the algorithm*	*Considered in the algorithm*

Table 3-3 Chapter summary

4 MIXED METHOD

The necessary data structures together with the underlying mathematical concepts to implement the balanced buffering algorithm have been studied. One major drawback of the proposed method is its relatively large runtime, as seen in section 3.9. To resolve this problem, a number of speedup methods will be suggested, the main one being a combination of the branch-and-bound and dynamic programming methods[1]. For more convenience, and because of its hybrid nature, the term *mixed method* is used to refer to the combination of branch-and-bound and dynamic programming. Also, two other speedup techniques will be presented which are somewhat related to the mixed method.

The first part of the work done in this chapter is a new approach of mixing branch-and-bound and dynamic programming techniques to improve the balanced buffering runtime. This idea comes from the fact that a remarkably high number of common sub-problems are encountered while solving a typical balanced buffering problem. As a result, and compared to the simple branch-and-bound method where no sub-problem solution is kept, the mixed method saves the sub-problem results and reuses them for quickly solving similar sub-problems. This allows faster search time with a reasonable memory footprint. While storing all sub-solutions would require much more memory than the simple branch-and-bound technique, the mixed method has shown only a small increase in memory usage during the tests. This is due to the use of dynamic programming which regroups similar sub-solutions.

Two additional speedup techniques will also be introduced for the first time, *smart bound* and *solution jumper*. Therefore, this chapter explains how the infrastructure is put in place for the mixed method, and how it is used to implement the additional speedup techniques.

Chapter Outline

Sections 4.1 through 4.4 explain how dynamic programming memory-reuse techniques are used to avoid redundant calculations and improve the buffering algorithm runtime. Section 4.1 illustrates the essential conditions under which dynamic programming and the branch-and-bound method can be

[1] Part of the work presented in this chapter was published in URSI ISSSE 2007 by [RABBANI and MAILHOT 2007].

combined. Section 4.2 presents the mixed method algorithm. Section 4.3 discusses the details of the mixed method implementation. Storing the sub-solution results can increase the memory usage. Section 4.4 introduces an efficient technique to reduce the memory consumption by detecting and deleting redundant solutions. The important issue of accessing solution lists is studied in section 4.5 where two new binary search trees are designed to meet the specific characteristics of the balanced buffering problem. Section 4.6 shows two additional techniques to improve the algorithm runtime. Finally, in section 4.7 the efficiency of the proposed speedup techniques is investigated.

4.1 Suitable Structure for Memory Reuse

4.1.1 Common sub-problems

In the previous chapter, a recursive structure was defined for solving the balanced buffering problem, where the solution found to a sub-sub-problem is independent from the solution to its originating sub-problem. In other words, given a source buffer and a number of sinks, there exists only one best buffer tree, regardless of buffering decisions made to the previous stages. This useful property somehow hides away the rest of the tree at the time of solving a sub-problem. Therefore it is possible to concentrate on finding the optimal solution of the newly generated sub-problem. If two different sub-problems share the same sub-solution, once the common sub-solution is calculated for one of them it can be used for the other sub-problem. Therefore, using this property can improve runtime. To have a better idea of how two buffering problems can share a common sub-problem, consider the example of figure 4-1. In this example two different buffers, *B1* and *B2*, are driving the same fanout number. If both trees use the same solution, i.e. *B3* at the root of sub-tree and a branching factor of 2, a common sub-problem is generated where *B3* is driving 3 fanouts.

Re-using the results of a solved problem is essentially what is done in dynamic programming, where redundant calculations are avoided by saving and utilizing the sub-problem solutions. As a result, it is useful to look into the basic structure of a typical dynamic programming problem and examine how to combine it with the branch-and-bound structure during balanced buffering.

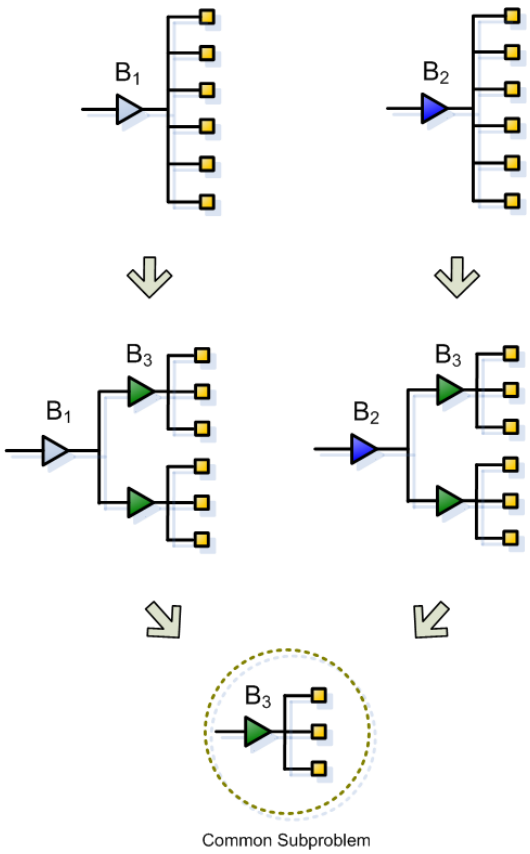


Figure 4-1 An example of common sub-problem

4.1.2 Branch-and-bound versus dynamic programming

Branch-and-bound methods can be very effective whenever independent sub-problems are encountered, where solving one does not affect the results of solving the others. They also need a tight bound in order to filter effectively the potentially enormous space of possible solutions. However, if the complete problem encompasses a large set of overlapping sub-problems, branch-and-bound methods can waste a significant amount of runtime, repeatedly recalculating the same sub-problems. In contrast, dynamic programming techniques can be applied whenever common sub-problems form the basic structure of a problem [BELLMAN, 1957] [MITCHELL, 1997] [KNUTH, 1998]. Recognizing an optimal substructure and the existence of overlapping sub-problems are two important properties of the problems which are solvable by dynamic programming. Using a structure tailored to dynamic programming, one can solve the problem efficiently in terms of both memory usage and runtime. The sub-problem graphs for branch-and-bound and dynamic programming are compared in figure 4-2. While an ideal structure for a problem to be solved by branch-and-bound is a graph tree where no sub-

problem overlapping occurs, dynamic programming method is very efficient when sub-problems are reused several times.

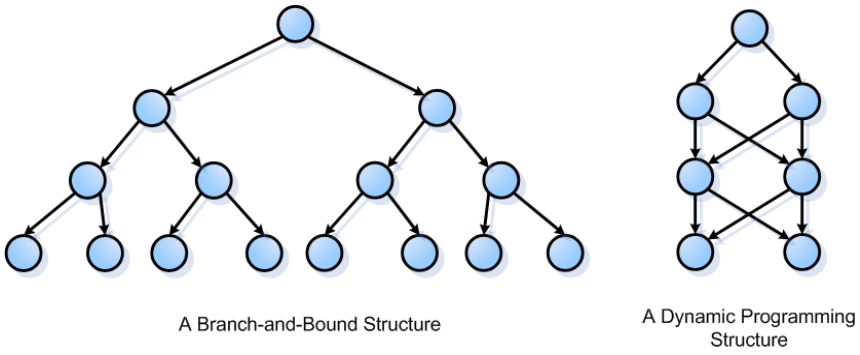


Figure 4-2 branch-and-bound vs. dynamic programming

4.1.3 Hybrid structure

In solving the balanced buffering problem, where there exist many common (overlapping) sub-solutions and where it is also possible to compute bounds, one can profit from the advantages of both branch-and-bound and dynamic programming methods. The idea of solution-reuse from dynamic programming can be applied to avoid unnecessary recalculations and when appropriate, utilize a bound to prune away non-promising sub-problems. The actual number of sub-problems visited during the search progress decreases with this method as solving similar sub-problems is avoided. This is symbolically shown in figure 4-3. In the left sub-problem graph some sub-problems appear and are solved more than once while they actually have the same solution. These are common sub-problems and they make the search space structure sub-optimal. A smaller search space is achieved by reusing the solution to common sub-problems when they are merged into one sub-problem. This is shown in the right sub-problem graph in figure 4-3.

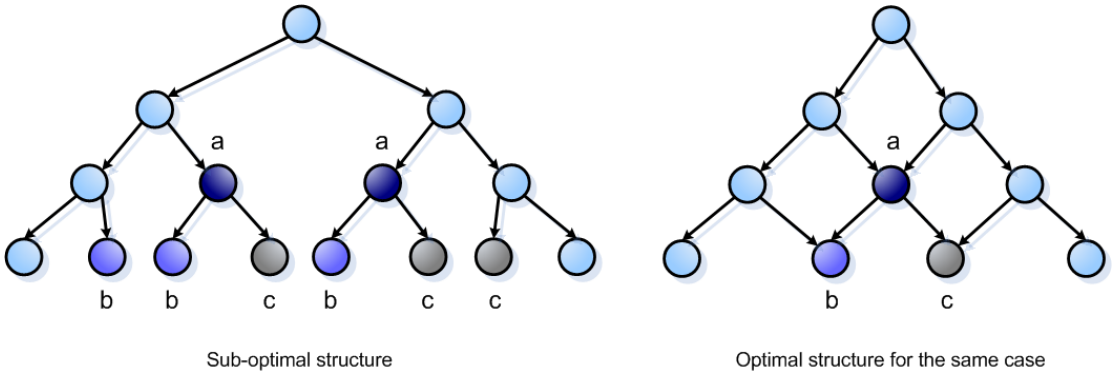


Figure 4-3 Search space structure for a balanced buffering problem

4.1.4 Conflict

There is an apparent contradiction between the ways branch-and-bound and dynamic programming techniques operate: branch-and-bound is essentially a top-down approach, whereas dynamic programming is a bottom-up one. This issue is resolved by traversing the search space in a top-down fashion, using branch-and-bound to limit the search as the traversal progresses. As end solutions are reached, dynamic programming takes over in order to propagate up the real best solutions encountered for every sub-problem. Consequently, the optimality of the branch-and-bound method is preserved while runtime is improved using dynamic programming. A general procedure of resolving the above conflict in two rounds is illustrated in figure 4-4. In this figure a common sub-problem (the dark node) is being shared by two higher level sub-problems. In the first round when this sub-problem is accessed for the first time (through the left parent node), it must be solved and its solution must be stored as there was no solution available for it before. After the common sub-problem finds its solution, the search traversal returns to the root (often a local one) and the common sub-problem is generated again through the right parent node. In this round, one can simply extract the solution by retrieving the results stored before and hence save time. If available, the solution to a sub-problem must always be propagated up to be used by the algorithm for solving higher level sub-problems in the future.

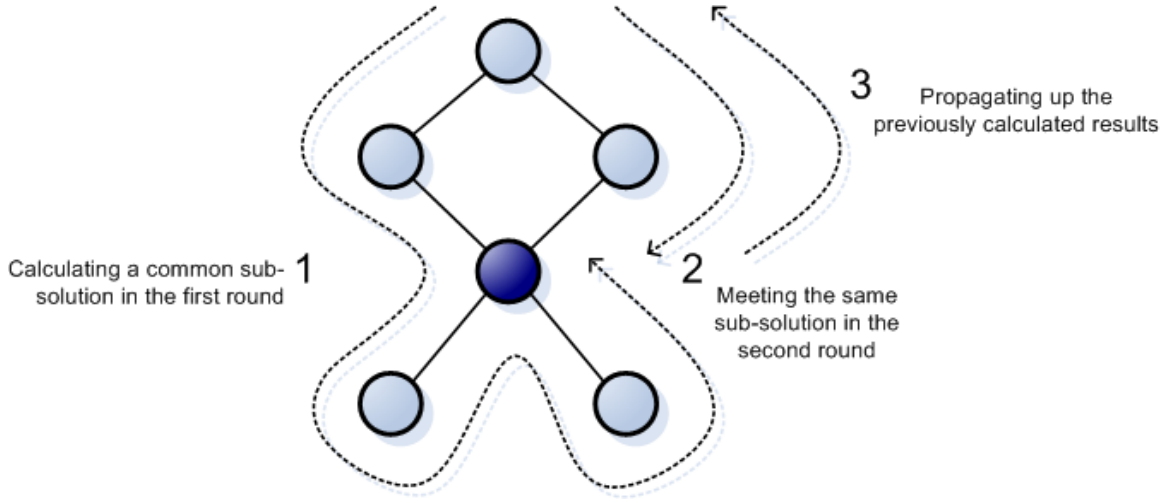


Figure 4-4 Combining two methods in solving a common sub-problem
(The common sub-problem is darkened)

4.1.5 Mixed method efficiency in solving balanced buffering

Due to the recursive structure of the sub-trees identified in the balanced buffering problem, many similar sub-problems are encountered. Therefore, one expects the mixed method to reuse many common solutions while searching for the best balanced buffer tree. As will be shown in section 4.7, sub-solution-reuse leads to better runtime, while keeping memory usage reasonably low. However, it is important to note that the efficiency of mixing the two techniques (branch-and-bound and dynamic programming) relies on 2 major factors:

1) The traversal ordering of the sub-problems
2) The structure of the search space

Underneath the first factor, two somewhat opposing goals are at work: runtime efficiency and memory usage. In order to optimally reduce memory, the best choice in solving a set of sub-problems would be to choose the one which leads to the largest sub-solution reuse. However, a comprehensive decision making algorithm would then be necessary and would negatively impact runtime. As a compromise, the sub-problems are currently traversed based on their ideal buffer tree delay, as discussed in chapter 3. This technique is generally satisfactory, as will be shown by the experimental results.

The second factor to consider is the way sub-problems are distributed in the search space. The number of sub-problems and the way their dependencies are established are directly influenced by the timing properties of buffers. In section 3.7 it was showed that the order of buffer arrangements in the best buffer tree obeys certain physical constraints. Due to the additional memory allocation needed for saving the solutions, the gain of using the mixed method must be carefully studied to guarantee a good payback.

The modifications to the balanced buffering method are now presented. They are necessary to implement the mixed method. In addition, this chapter covers a specific binary search tree introduced and designed to help the runtime of the mixed method. Because the sub-problem solutions are saved in the mixed method, memory usage has to be dealt with, a concern which does not exist in the simple branch-and-bound algorithm. One efficient technique will be introduced to handle the memory usage problem. Moreover, two new speedup methods will also be presented to effectively reduce the runtime. The efficiency of such modifications and techniques will be studied in section 4.7.

4.2 Mixed Method Algorithm

The mixed method flowchart is provided in figure 4-5 explaining the modifications done in the main balanced buffering algorithm. More details about the techniques put in place to improve runtime and memory consumption follow in the next sections.

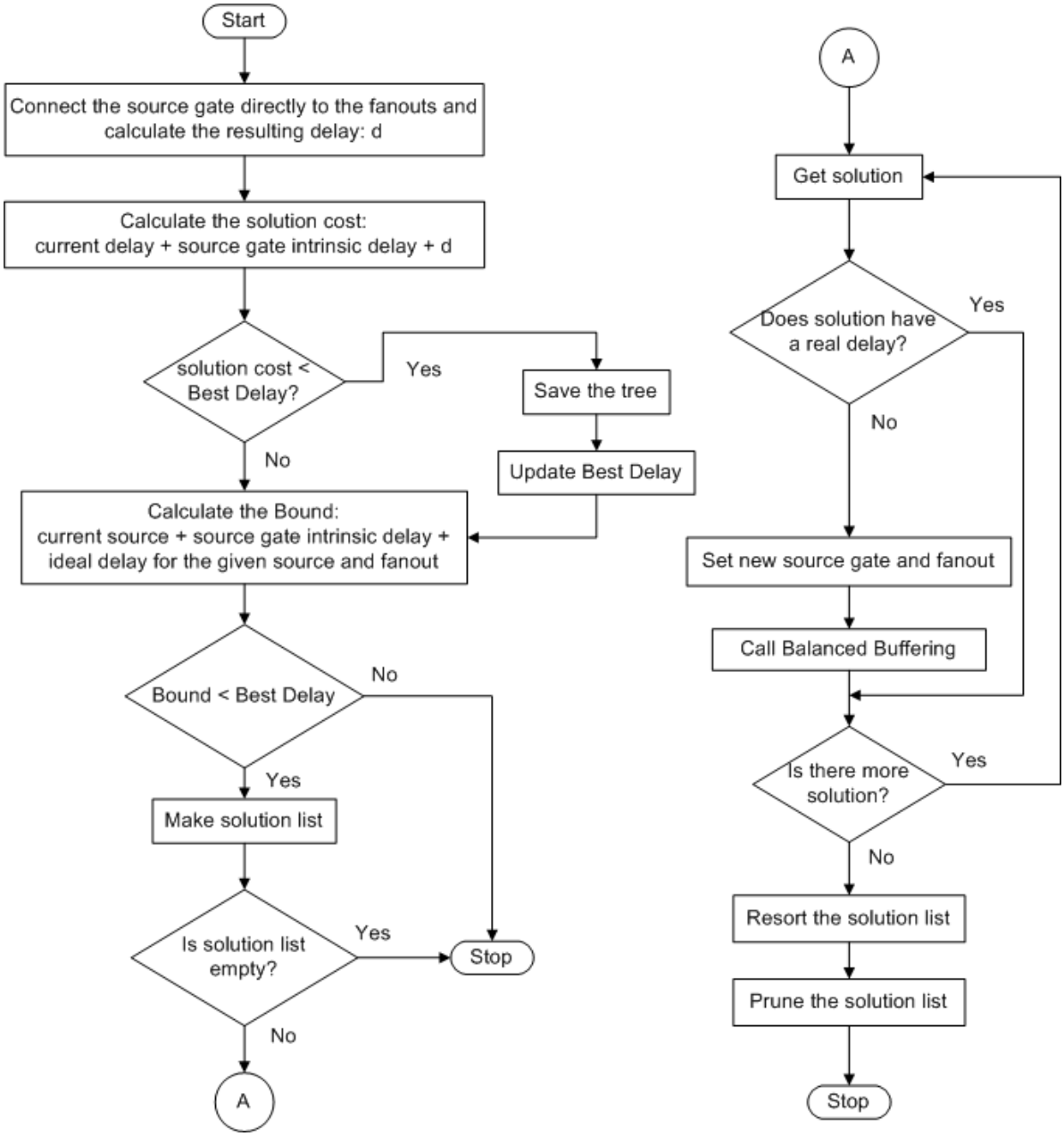


Figure 4-5 Mixed method flowchart

4.3 When Should Memory Reuse Be Performed?

With dynamic programming, it is necessary to keep the sub-problem results for further reference. Therefore, solution lists are updated whenever solving a sub-problem yields a practical solution. A practical solution, a completely solved sub-problem, is one that never fails to pass the bound during the solving process. The outcome of completely solving a sub-problem is getting the best buffer tree for that sub-problem. It is only achieved when the search process has successfully terminated without any interruption by the bound, where a negative phase buffer tree can never be a practical solution. Thus, a sub-problem solution can be saved whenever its best buffer tree becomes available.

The saving and memory-reusing procedure must be handled at different stages. Those steps include either updating the delay values of the solutions or reusing them to calculate some others. The major operations of the memory reuse system are:

1) **Making Basic Solutions**. There always exist a number of empty solution lists in the search space, which correspond to single fanout problems. The only solution to that class of sub-problems is connecting the source buffer directly to the sinks, without any intermediate buffer insertion. To make those solutions reusable, a no-buffering solution is simply added to the empty solution lists and it is called *basic solution*. The delay of a basic solution is quickly calculated and saved together with the solution, as there is no better delay expected from such a solution. Hence, the basic solutions are the initial points where memory re-use starts. In figure 4-6 it is shown how basic solutions must be produced and used.

2) **Making Solution Lists.** To make memory reuse effective, one has to first search for previously calculated solutions at the time of making a new solution list. As the solutions are enumerated, the search space is examined for any solution result already available. This is the core of the speed enhancement: most of the enumerated sub-problems find their complete solutions at the time of making the solution list, before the list even being explored by the main algorithm. In addition, in order to have an effective search traversal in the main algorithm, the solution list is sorted in terms of minimum delay using both real and ideal delay values. In the sorting process and if available, the real delay values of the solutions are used, and if not available their ideal delay values are taken into account.

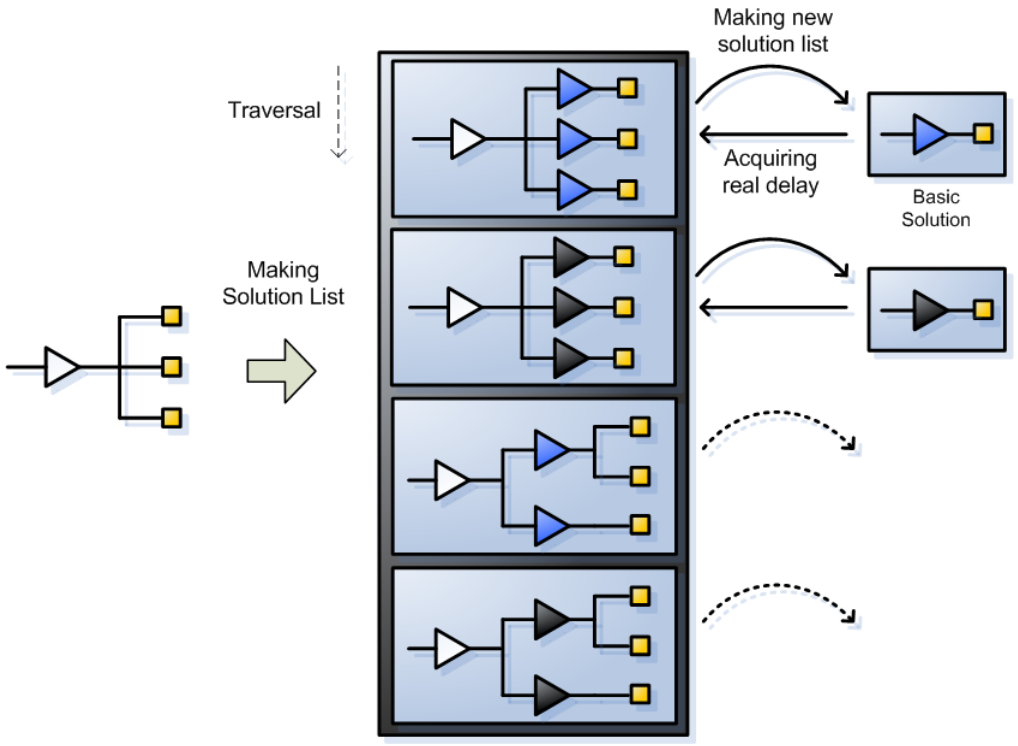


Figure 4-6 Basic solutions application

At the end of the sorting step, if a complete solution (with a real) delay places at the top of the list, the solution list will be marked as *complete* and its best solution will be later used to extract the best real delay. Otherwise, the solution list will be marked as *incomplete*. The set of solutions with no real delay that belong to an incomplete solution become candidate to be solved by the *balanced buffering* algorithm. Note however that a solution list whose ideal delay is larger than the best real solution will remain incomplete as it will not be processed. An example of making a solution list is provided in figure 4-7 where solutions are named *a* through *g*. All solutions have only ideal delay at the beginning. This is shown by *check* marks in the middle column of the list. Then the search space is verified for any solution previously calculated. Those solutions that have been calculated before and have a real delay are updated in the list. This is shown by check marks in the right column. Then the list is sorted based on real delay values, and if not available, based on ideal delay values. For example if solution *b* has a greater real delay than the ideal delay of *c*, it will be placed in a lower position in the solution list.

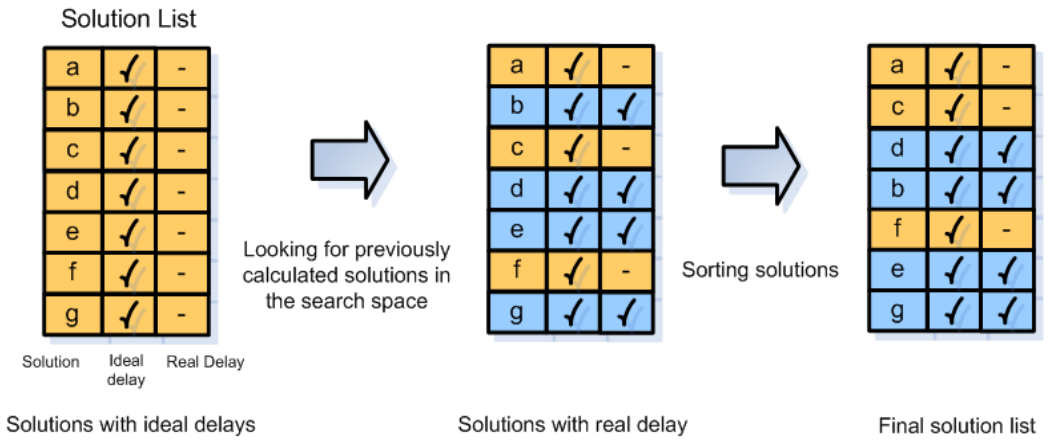


Figure 4-7 Setting real delays and sorting the solutions when making a solution list

3) **Updating the Solution Delays.** Any buffer tree will only contain basic solutions at its deepest buffering stage, i.e. the last level of buffers before the fanouts. For example, if a buffer tree has 5 levels of buffers, the 5th level contains only basic solution as the buffers are directly connected to the fanouts. Any sub-problem at the previous level can easily use the real delay values from the basic solutions. This propagates back the real best solution to the preceding stages, repeatedly to the root of tree. This is where the algorithm performs like a dynamic programming method. Therefore, the result of solving a sub-sub-problem may be used by the originating sub-problem if and only if it provides a *complete* solution list. This is where the algorithm varies from a pure dynamic programming method. However, the proposed method differs from pure dynamic programming in that it uses a bound to eliminate some of the sub-solution calculations.

   Once the balanced buffering algorithm finishes fetching and solving the sub-problems of a solution list, the list must be resorted as some incomplete solutions may now have real delay values. In the end, the first solution of the list, whether having a real delay or not, is returned to be examined by the originating sub-problem. The returned solution is then used to update the delay field of the sub-problem it is derived from. This occurs after the balanced buffering algorithm returns from solving a sub-problem. The procedure of updating the real delay value of a solution is shown in figure 4-8. First sub-problem *a* is encountered in a solution list. Then it is solved by enumerating and exploring its solution list. Finally when all solutions in the generated solution list have obtained their real delays, the list is sorted and the best solution,

which is the one with minimum real delay, is returned. This is when the real delay of solution *a* is updated.

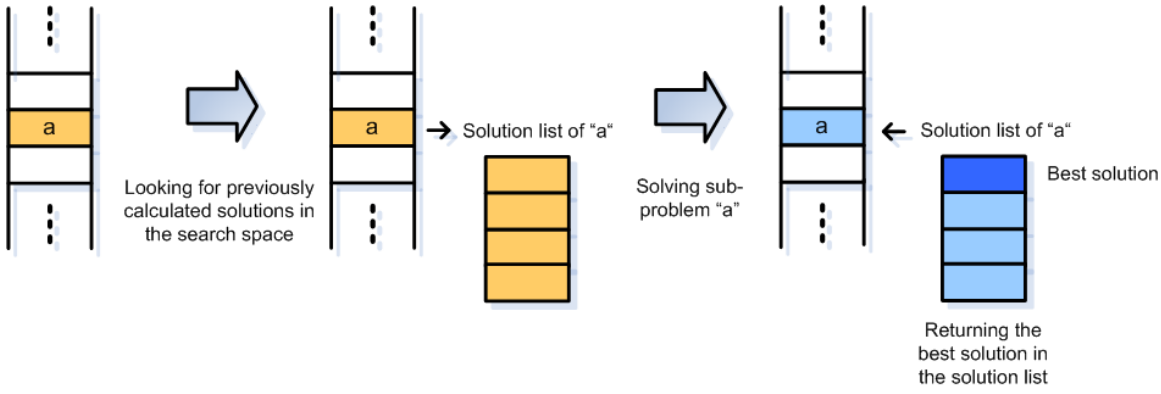


Figure 4-8 A sub-problem being updated

4.4 Solution List Pruning

There is often a trade-off between memory usage and runtime. Whereas the runtime of balanced buffering is improved by saving solution lists, a higher amount of memory is required in comparison with a simple branch-and-bound system. To resolve this drawback and help keeping the search space small, solution lists are pruned by deleting those solutions which are dominated by a complete solution. If the ideal delay of an unsolved solution is already worse than the real delay of a complete solution in the same list, it can never be better than that complete solution, and hence becomes redundant. Pruning must be performed during solution list enumeration, and after the solutions are sorted with respect to their delay values.

4.4.1 Pruning the positive-phase solution lists

Positive-phase solution lists are those which are generated for positive-phase balanced buffering sub-problems (see section 3.8). To prune a positive phase solution list, where there is a mixture of sorted solutions with ideal and real delay, the solution with minimum real delay is found (if there is any) and the rest of the list is deleted. The example in figure 4-9 shows the procedure. According to the list, solution *c* is the best solution with real delay and the inferior solutions can be discarded. Although solutions *e* and *f* are not complete, they have to be discarded as well. This is due to the fact that these two solutions have already a larger estimated delay than solution *c* and even if they are solved they will not get any smaller delay value.

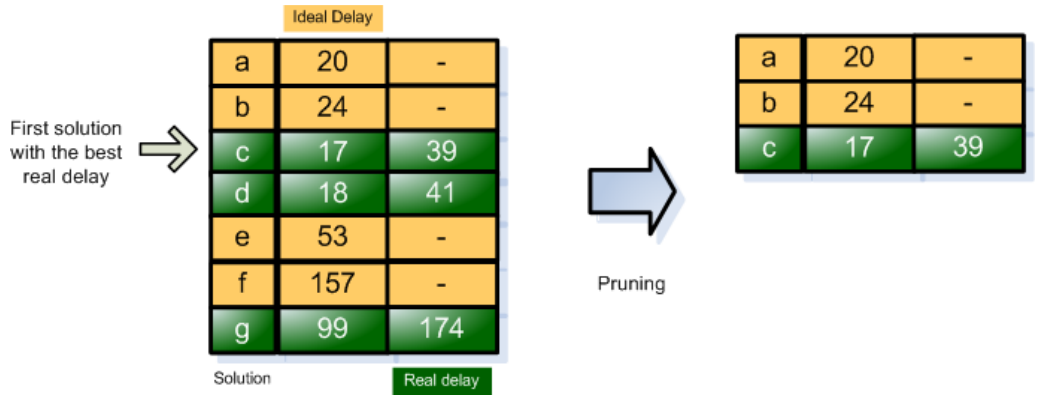


Figure 4-9 Pruning a positive-phase list

4.4.2 Pruning the Negative-Phase Solution Lists

Pruning a negative-phase solution list is somewhat more complicated. There are two sub-lists (positive-phase and negative-phase) which are sorted separately and therefore, it is not possible to just use the first real solution to prune away the combined list. Whereas a simple way to do the task is to prune each sub-list independently, the best real solution of every sub-list is used to improve pruning of the other one. The following steps must be taken in order to effectively prune a negative-phase solution list:

1) Sort each sub-list in terms of minimum delay
2) Find the best real solution of each sub-list
3) Between the two best real solutions select the one with smaller real delay
4) Add the selected solution to the sub-list it does not belong to
5) Resort the sub-list in which the selected solution has been inserted
6) Prune each sub-list

Note that by doing so, a solution with inverting buffer may be used to prune a non-inverting sub-list and vice versa. However, the pruning technique applied does not violate the primary assumptions to have two independent sub-lists in solving a negative-phase sub-problem, as the added solution is ignored by the main algorithm due to its real delay.

The example in figure 4-10 clarifies the multi-step pruning technique. The same procedure can be applied to the case in which the non-inverting sub-list has the best real solution. In order to minimize memory usage, the pruning function is applied at the end of a solutions list exploration in the main algorithm, as some solutions might now have acquired a real delay value. Thus, a solution list can

always be kept at its minimum size. The tests show that pruning a solution list has an effective role in reducing memory usage of the algorithm, as will be shown in section 4.7.

Figure 4-10 Pruning a negative-phase list

4.5 Search Space

To summarize what has been described thus far about the solution management and reuse in the search space, they are categorized into three major groups:

a) Eliminated redundant solutions: those solutions are cut by the bound. Their solution lists are not built.

b) Complete solutions: those solutions whose results are already available and are built of previously calculated sub-solutions. Their sub-solutions are not recalculated; instead, the best solution in their solution lists is used to avoid redundant calculations.

c) Candidate solutions: those solutions which have both complete and redundant solutions in their solution lists. Their solution lists are saved anyways, although it is not possible to use them to obtain the best solution. This helps them being solved faster during further references.

These 3 solution types are shown in figure 4-11 where a candidate solution has a complete and a filtered solution in its solution list (solution *a* and *b* respectively).

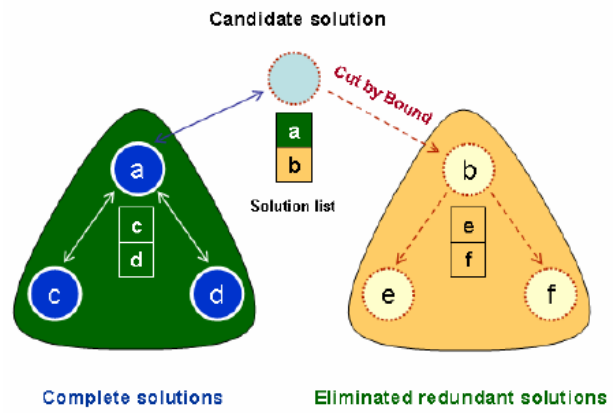


Figure 4-11 Solution types

4.5.1 Saving and maintaining the solution lists

For solution list reuse to be useful, an efficient retrieval system is needed. Every unique sub-problem is defined by its source buffer and the number of fanouts it is driving. Therefore, a solution list can easily be found using these two values. As a consequence, the feasible space of the solution list lies in a 2-dimensional coordinate system in which each point represents a solution list with the given pair of buffer type and fanout number. If there are *n* buffer types and *m* fanouts, *m*n* distinct points will be available in the coordinate system to be used as the identifier (access key). This is shown in figure 4-12.

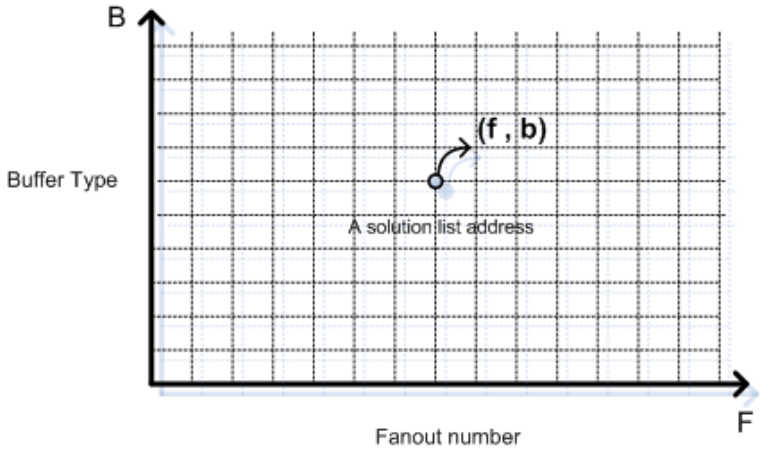


Figure 4-12 2-dimensional coordinate

An access key structure is symbolically illustrated in figure 4-13.

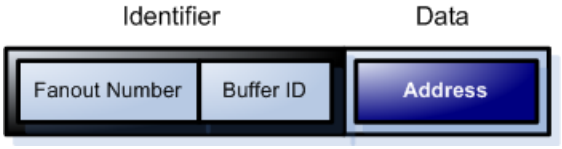


Figure 4-13 Solution list access key

To have the best effect, an appropriate data structure should be carefully chosen to best harmonize with the properties of the typical balanced buffering problems. Some common look-up methods along with the related data structures commonly used in similar applications are going to be studied. A new data structure will then be introduced that is well-tailored to the nature of the balanced buffering algorithm.

4.5.2 Look-up table

A 2-dimensional look-up table can easily solve the solution list access problem. However, despite its simple structure, its implementation is not straightforward. There are two common ways to implement a look-up table; one is to construct a static table using static arrays, the other one is to have a dynamic table consisting of dynamic arrays. There is a crucial tradeoff between those two methods: if static arrays are used, the access time is constant, but the memory pre-allocated to save the solution lists can be very high for large search spaces. In contrast, if dynamic arrays are used an efficient memory usage

is put in place, but on average, poor access time is observed. This is because the solution lists are generated randomly and hence the look-up table can not have any specific key ordering to make constant access time possible. In fact, one has to implement a dynamic table as a linked list, which yields a linear access time O (n) on average.

4.5.3 Other Methods

Two other widely used methods are hash tables and binary search trees. While a well-implemented hash table can result in an acceptable memory usage and a good access time, finding an efficient hash function becomes very challenging when a 2-dimensional key must be mapped into address values. If poorly chosen, the hash function can badly impact the length of the linked list sharing the same hash value, which eventually results in high access time.

As an alternative, a binary search tree can yield good access time (O (log n) in general) while its implementation does not demand any complex mathematical basis. In fact, the class of self-balancing binary search trees guarantees a good worst-case runtime of O (log n) for the basic operations such as insert, search and delete. That group of binary search trees may seem suitable for the balanced buffering problem, meeting all design objectives such as acceptable search time and good memory footprint.

On the other hand, if the distribution of solution lists usage number for a typical balanced buffering problem is carefully examined, an uneven distribution will be observed, having more a Zipf's law [LI, 1992] than a uniform distribution. This can be explained as some common sub-problems, e.g. those with a fanout number of 2, are more popular than the others, and hence their total reference numbers appear to be higher than non-popular ones, say those with large fanout numbers. This is clearly observable in figure 4-14. In this figure the usage number graph for the first the 20 sub-problems of a buffering problem with 86 sub-problems is presented. The total fanout number is 1000 and a buffer library with 6 non-inverting buffers is used. The properties of the first 20 sub-problems encountered in this example are tabulated and shown in table 4-1. Sub-problems are sorted in terms of usage number.

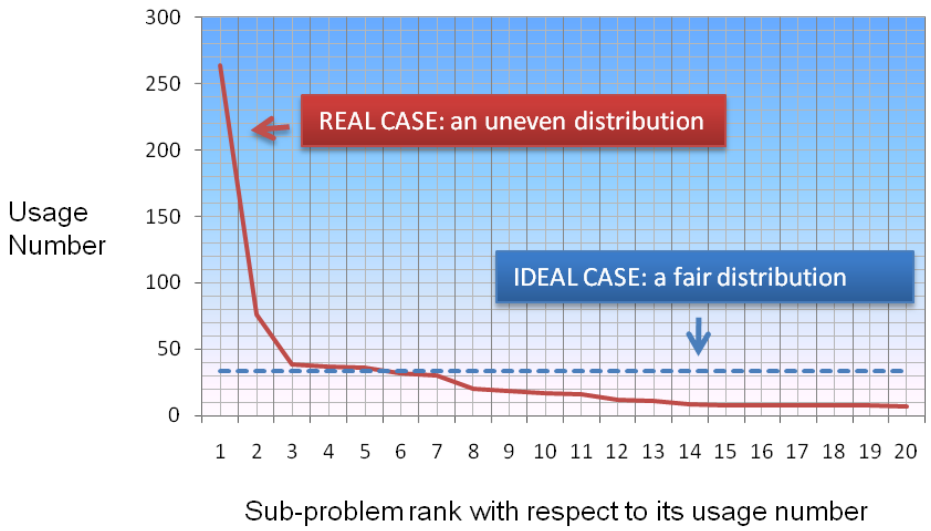


Figure 4-14 Usage number distribution for the given example

Sub-Problem	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Source Buffer	B_6	B_4	B_6	B_4	B_6	B_6	B_6	B_6	B_4	B_6	B_6	B_2	B_2	B_2	B_6	B_6	B_6	B_6	B_6	B_6
Fanout Number	2	2	4	3	7	3	5	21	5	6	14	7	2	21	10	11	13	16	25	9
Usage Number	264	76	39	37	36	32	30	20	19	17	16	12	11	9	8	8	8	8	8	7

Table 4-1 Properties of the first 20 most common sub-problems

This characteristic of the balanced buffering problem suggests a more dynamic data structure. It is therefore very helpful to have a self-organizing data structure capable of reshaping itself such that the access time of popular keys reduces as their usage number increases. To that effect, a binary search tree must be modified such that the keys of the more common sub-solutions are found closer to the root. This is achieved through a dynamic binary tree which gradually pushes up the popular sub-solutions. Two different classes of such self-organizing binary search trees have been invented and implemented in this thesis, called *lazy weight binary search tree* and *perfectly balanced binary search tree*. They are going to be explained in details in sections 4.5.4 through 4.5.7 and their impact on the algorithm runtime will be studied in section 4.7.

4.5.4 Modified binary search tree

In this section, a simple binary search tree is modified such that the solution list addresses are stored and accessed efficiently. In a typical binary search tree each node is assigned a key and contains a value as data. The binary search tree must hold the binary search tree property, which is the following:

> **Binary Search Tree Property:** *Let x be a node in a binary search tree. If y is a node in the left sub-tree of x, then* ***key[y]*** $\leq$ ***key[x]***. *If y is a node in the right sub-tree of x, then* ***key[x]*** $\leq$ ***key[y]***. [CORMEN et al, 2001]

In a simple binary tree only one key is needed to find a node. However, in the balanced buffering case where the access key is a pair of values (fanout number, buffer type), it is necessary to add an additional dimension to each node to make it possible to implement a 2-dimensional access key. This is done by inserting an additional data structure inside each tree node. There are therefore two different keys: one general key to find the correct binary tree node, and a secondary one to extract the data from the encapsulated data structure inside that node.

Given the fact that the number of buffer types in a buffer library does not normally exceed a few dozens, a suitable option is to use buffer types as the secondary key. Hence, if there are **B** buffers in the buffer library, the encapsulated data structure can be a relatively small look-up table with a maximum of **B** rows. Using a static look-up table is a brute force way. It usually yields a fast local access time and a reasonable memory usage. On the other hand, the fanout number is used as the general key of the binary search tree. An example of such a data structure is shown in figure 4-15, where a set of fanout numbers ($f_1 \ldots f_{14}$) is used as the general key and a set of buffer types (0…B-1) is used to provide local access to the look-up table. A solution list address is then found by (f , b) where f is fanout number and b is the buffer type.

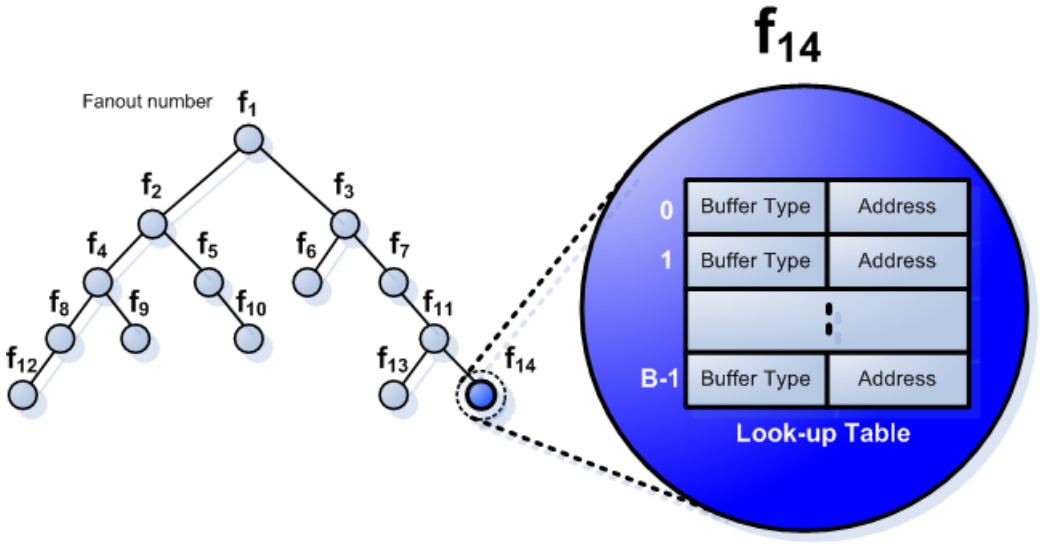


Figure 4-15 A binary search tree with look-up tables inside each node.

Solution lists are generated randomly. This means there is no specific order for their address to appear in the look-up table. Therefore, if a dynamic table is being used one has to search the whole table for

the requested address simply because there is no way to calculate its position. This effort makes the time complexity of local access to the solution O (b) where b is the buffer library size. As mentioned in section 4.5.2 a static table can yield a constant access time, but it is often very sparse and hence wastes memory. A useful technique to reduce the size of the embedded static look-up tables is to combine static and dynamic tables. This is explained by the following example.

Consider there are **B** buffer types in the buffer library. Therefore a simple static look-up table using the buffer types as its access key would have **B** rows. To decrease the number of empty rows, dynamic arrays are used to build up the table, and insert empty slots only when it is necessary. To see how the method works, consider the example of figure 4-16. In this example different buffer types are assigned a unique number. At first, there are only two slots in the table (left table) whose buffer types have consecutive numbers. If a sub-problem with the same fanout number but with a different buffer type, say type 5 is processed, its solution list address must be saved in the 5th row in order to keep the table in order. The gap produced between the second and the fifth rows must then be filled with empty slots. This is similar to a static table in the sense that one can easily compute the position of a requested address in constant time. Since the number of inserted empty slots is usually less than the number of empty rows in a sparse simple static table, less memory is consumed while the access time remains constant.

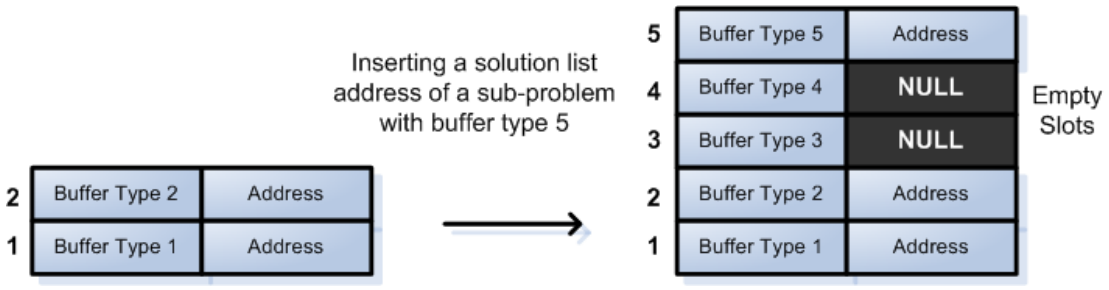


Figure 4-16 Filling up a dynamic look-up table

Note that the buffer type labeling strategy can play a major role in decreasing the number of empty slots. If those buffer types which are more frequently used by the algorithm are somehow identified and labeled with low numbers, definitely fewer gaps in the look-up tables will be encountered.

4.5.5 Lazy weight binary search tree (LWB)

A LWB is a binary tree in which each node is assigned a weight value. It slowly modifies its structure as the program progresses, such that after a certain amount of time the access time of highly

referenced solution lists reduces. The basic structure required for a LWB was proposed in section 4.5.4. In this section a method will be put together to enable a tree to be self-organizing. The purpose of using the term *lazy* to call this specific type of binary trees will be explained after introducing the architecture of the search tree.

The goal is to dynamically restructure a binary tree such that highly shared common sub-problems can be accessed faster than the others. It is then required to measure the usage number of every sub-problem. To this end, a data field is added to every solution list and is incremented whenever the solution list is referenced. This usage number is then used during the tree self-reorganizing operations. The weight of a node in a binary search tree is defined as:

$$Node_{weight} = \sum_{i=1}^{j} Solution\ List_i \rightarrow usage\ number$$

Where j is the number of available solution lists and *Solution List$_i$ -> usage number* means the usage number of the i_{th} solution list. Next, the self-reorganizing rule is defined as follows:

> **Self Reorganizing Rule:** *The weight of a child node must be less than, or at most equal to the weight of its parent node.*

A function is also defined to send up the more accessed nodes, called *push-up*:

> **Push-up Function**: *while the current node weight is larger than its parent weight, push it up by a left tree rotation, if it is the right child of its parent, and a right tree rotation, if it is the left child of its parent.*

Note that tree rotation is used rather than simply swapping two nodes. This is due to the fact that the key orders *are not* subject to change, only the node weight orders. By tree rotations instead of node-swapping the binary search tree property is preserved.

The push-up function works as follows: once a solution list is created or referenced, its usage number is incremented. Therefore the total usage number of the corresponding node is incremented. If required, push-up does an adequate number of tree rotations to send up the node that has been recently accessed. This procedure continues until the self-reorganizing property holds again. An example of how the push-up function works is shown in figure 4-17. A node with a key value of 20 and a usage number of 62 is accessed. As a result, its usage number becomes 63, which violates the self-

reorganizing rule. From here, push-up takes over and performs a right tree ration. In this case, only one tree rotation is enough.

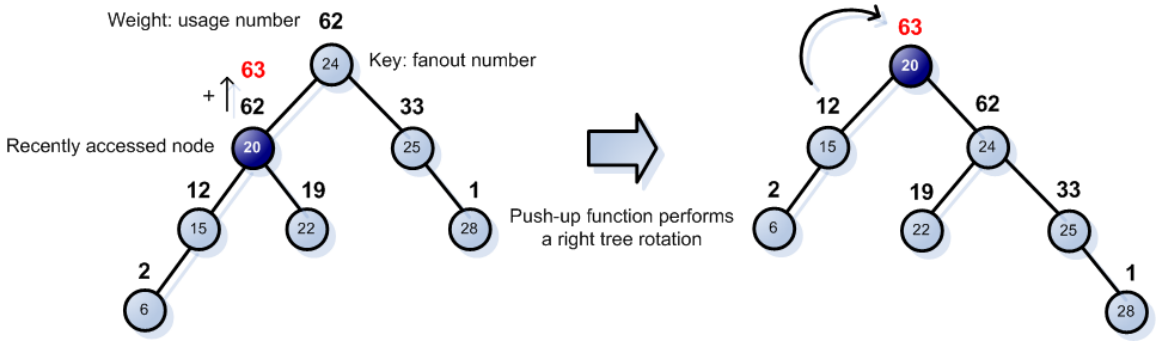


Figure 4-17 An example of push-up function.

Push-up plays a tree maintenance role which preserves the self-reorganizing property of the proposed search tree. There is also a subsidiary advantage of the previously presented 2-dimensional data structure. When a sub-problem is visited for the first time in the search pace and has a fanout number leading to a highly used node, it can profit from a good access time by being initially placed near the root. This is because that node was pushed up before and now lives somewhere close to the root. This also helps stabilizing the tree restructuring process by effectively decreasing the number of tree rotations.

Sometimes a chain of equally weighted nodes appears in the search tree. Although it is not often the case, it can cause a particular problem. If the last node of such a series is accessed, its usage number is incremented. Having a node weight heavier than its parent, the push-up function will send it up right to the top of the chain. Now as they were all equally weighted before, one can suppose that the probability of being accessed for all of them is roughly equal, and consequently another node of that chain will soon be used. If this occurs, and the node is unluckily found at the tail of the chain, the push-up function has to repeat the procedure to send it up. In that case, one spends a lot of time adjusting the tree, where the time gained from the self reorganizing method becomes insignificant in comparison with the time spent in manipulating the chained nodes. This is illustrated in figure 4-18 by an example. In this figure node *a*, with a usage number of 6, is accessed and has to be sent up to the root. An access to any node other than node *a* triggers a series of time consuming tree rotations. As a compromise, a less sensitive function is introduced which is called *move-once*.

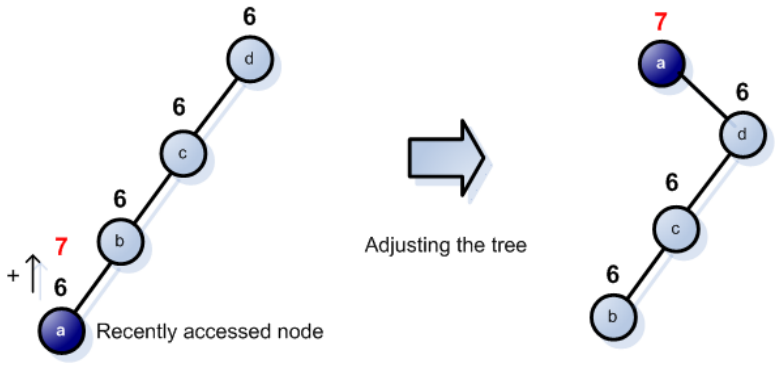


Figure 4-18 Push-up operations for a chain of equally weighted nodes

Move-once is similar to push-up except that it performs a maximum of one rotation at a time. It is defined as follows.

> **Move-Once function**: ***if*** *the current node weight is larger than its parent weight, push it up by a left tree rotation, if it is the right child of its parent, and a right tree rotation, it is the left child of its parent.*

Note that the only difference between the move-once and the push-up definitions is the *while* condition replaced by the *if* condition. In most cases, move-once and push-up both perform at most one rotation when a solution list is accessed, if the referenced node does not belong to any chain of equally weighted nodes. If it does, move-once moves the node only one level upward and by doing so it waits for further references to that specific node. If that node contains highly used solution lists, it soon reaches the head of the chain and finds its proper position. If not, time is saved by postponing the tree adjustment. In practice, the loss of slowly pushing up a node belonging to a chain of equally weighted nodes is negligible as the length of such chains is generally short. Due to its laziness in self-reorganizing, this type of data structure has been called *lazy*.

4.5.6 Perfectly balanced binary search tree (PBB)

In this section a second type of self-reorganizing binary search trees is presented, called *perfectly balanced binary search tree*. While the first type has demonstrated good result in tests, a different data structure is introduced to provide the underlying concepts for future extensions of the solution lists access method. An ideal search tree would set the distance of each node such that the sum of all node

access probabilities is maximized. If there exist **n** nodes in the search tree, one can consider an n-variable function for which a maximum probability value is found:

$$\text{Maximize } F \text{ where:} \qquad F = \sum_{i=1}^{n} Solution\ List_i \rightarrow access\ probability$$

Where n is the number of available solution lists and *Solution List*$_i$ *-> access probability* means the access probability of the i_{th} solution list. The optimal structure presenting a maximum value for F is only achieved by global decisions upon the tree structure. However, this is not achievable by LWB, where each re-organizing decision is merely based on local optimization. In order to design a self-reorganizing method with an optimal node distribution, four tree properties must be defined:

Property 1: *The weight of a node is the sum of its left sub-tree total weight, plus its right sub-tree total weight, plus its own usage number. This is shown in figure 4-19.*

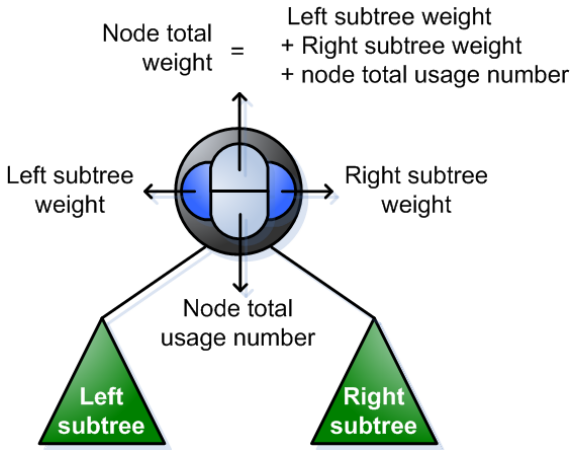


Figure 4-19 The structure of the defined node

Property 2: *The applied search tree is an extended binary tree where each regular node has exactly two children. The null sub-trees are replaced with special* ***Null*** *nodes that do not have any children.*

Property 3 (Balancing Property): *Consider figure 4-20:*

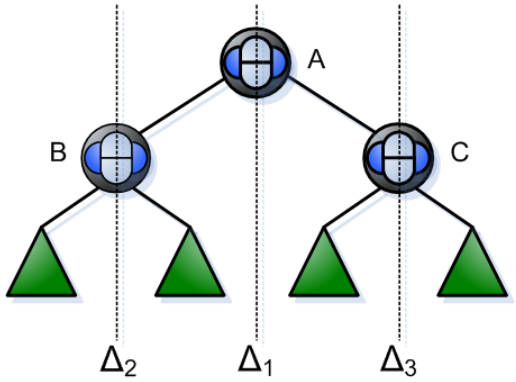


Figure 4-20 Defining balancing property

Δ_1, Δ_2 *and* Δ_3 *are defined as follows:*

$$\Delta 1 = |LTW - RTW| \quad (4\text{-}1)$$

$$\Delta 2 = |LTW - (RTW + AUN + CW)| \quad (4\text{-}2)$$

$$\Delta 3 = |RTW - (LTW + AUN + BW)| \quad (4\text{-}3)$$

Where LTW is the left sub-tree weight, RTW is the right sub-tree weight, AUN is the usage number of node A, CW is the weight of node C and BW is the weight of node B. Please note that there is a difference between weight and usage number, as weight is the sum of usage numbers of all sub-trees of a node, while the usage number is limited to the node itself. One now claims the tree is perfectly balanced if and only if:

$$\Delta_1 = \min\left(\Delta_1, \Delta_2, \Delta_3\right)$$

Proof of Property 3. It is trivial that if Δ_1 is not minimum, either of the children becomes the parent, any of which that has smaller Δ.

Property 4: *Every sub-tree in a perfectly balanced binary search Tree must also be perfectly balanced.*

Holding all defined properties, the whole search tree presents an efficient structure in terms of access probability. The tree with such an attribute is then called a perfectly balanced binary search tree (PBB). To implement a PBB, two functions must be defined: *see-saw-up* and *see-saw-down*.

See-saw-up is used to move up a node in 6 steps:

1) Consider a triple of nodes, consisting of current node, its sibling and its parent.
2) Calculate Δ for each the current node and its parent, using property 3.
3) If the calculated Δ for the current node is smaller than the calculated Δ of its parent, move it up by a tree rotation. If not, the function terminates.
4) Update the weight fields of each node
5) Do a see-saw-down for the new child (explanation follows)
6) Repeat the steps of 1 to 3 for the recently pushed up node.

An example of the see-saw-up function execution is provided in figure 4-21. A node is accessed (darkened) and since property 3 is violated the tree must be re-organized. Note that both node weight (220) and node usage number (147) have been increased. Also, when the former parent becomes a new child, its weight must be updated by summing up its own usage number, its left and right sub-tree weights.

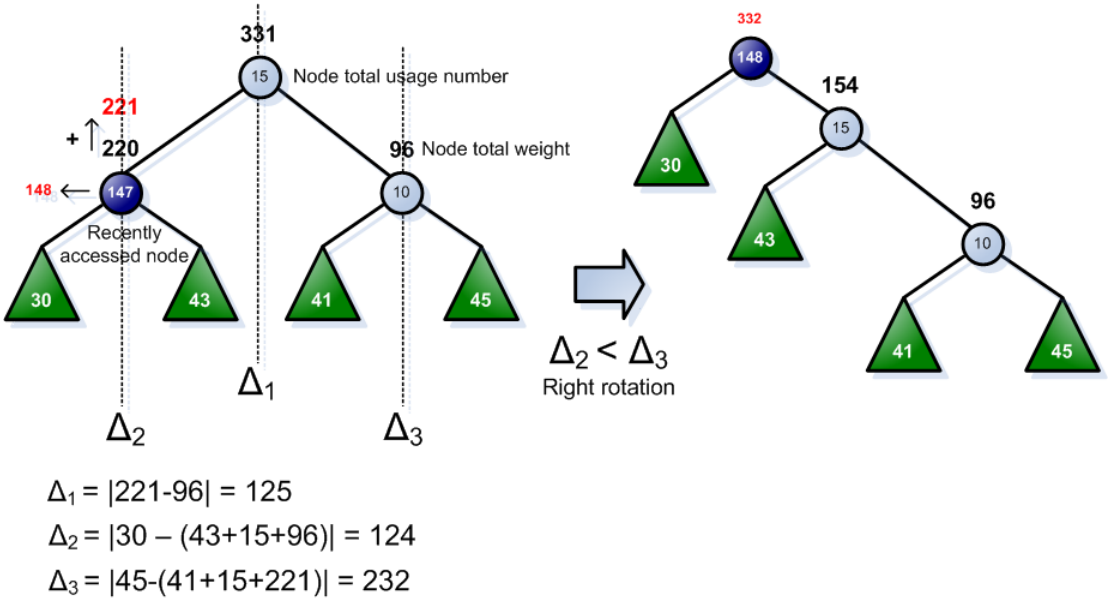


Figure 4-21 Example of see-saw-up function.

Each time see-saw-up performs a tree rotation, the former parent node becomes the root of a new sub-tree. Therefore, the property 4 must be verified to be held for this new sub-tree. If the mentioned property is violated, then a number of operations must be done in order for the sub-tree to retrieve its perfectly balanced attributes. These operations are summarized in a function called see-saw-down. See-saw-down is used to push down a node in 6 steps:

1) Consider a triple of nodes, consisting of the current node, and its two children
2) Calculate Δ for each node, using property 3.
3) Find the minimum $\min(\Delta_1, \Delta_2, \Delta_3)$
4) If the minimum Δ does not belong to the root of the sub-tree, rotate the tree such that the node with minimum Δ is placed at the root. If not, the function terminates.
5) Update the weight field of the new root and the new child
6) Repeat the steps of 1 to 4 for the recently pushed down node.

Reorganizing the binary tree requires a combination of see-saw-up and see-saw-down functions. For each see-saw-up it is required to do a number of see-saw-down to hold the tree properties. Although this could become a drawback for the method, the experimental results have shown that a good runtime is achieved in practice. This is because of the efficient structure of the search tree which provides fast access times to solution lists. This section ends with a potentially better method using the two types of search tree introduced.

4.5.7 Hybrid Method

The PBBs are very sensitive to any change in solution list usage numbers, and may trigger a series of tree rotations at a single solution list access. In contrast, the LWBs are less sensitive and take a minimum number of actions to modify the tree. One can profit from both properties by carefully choosing either of these methods for different phases of the search space exploration.

During the initial stages of balanced buffering, the tree is small and is subject to a lot of changes. A sensitive method could then waste a lot of time at this point. On the other hand, after a certain period of time and once an adequate number of solution lists are created and used, the structure of the search tree becomes more stable. At this point, due to a relatively large search space, the access times become more and more crucial. As a result, a lazy access method may be applied in the first steps of a search tree evolution, and a more sensitive but efficient method is used when the search tree is stabilized.

During the tests, a particular number of solution lists references are chosen randomly, say **m**, to indicate the time of switching the tree manipulation method from lazy to sensitive. LWB is then utilized to control the search tree construction before **m** references occur, and PBB is used over the second phase. In general, applying the proposed solution resulted in an overall runtime improvement in the program. However, the main difficulty with successfully implementing such a method is to find a proper **m**. Based on the experimental observations, the author believes that there is an optimum **m** for which one obtains the minimum runtime for the presented buffering algorithm. **m** is symbolically shown in figure 4-22 to better explain the idea.

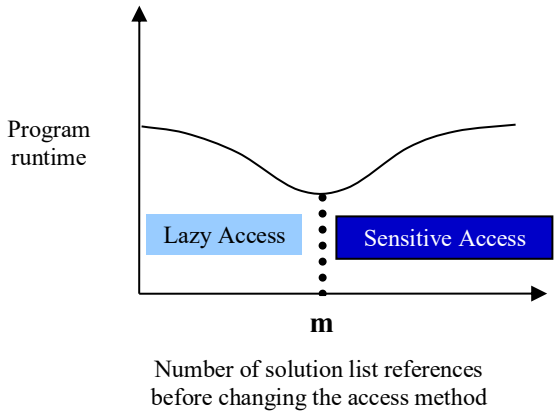


Figure 4-22 Existence of a possible ***m***.

The number of appropriate references before altering the tree access method is directly influenced by how the traversal in the search space progresses. The sooner the highly used solutions are processed, the smaller **m** should be needed, as the search tree is stabilized sooner. As a result, an appropriate value for **m** could be found by somehow pre-evaluating potential sub-problems of a balanced buffering problem, which is far beyond the scope of the research in this thesis. Therefore, it is left for future studies to find a function capable of calculating a good approximation for **m**.

4.6 More Speedup Techniques

In this section, two more speedup techniques are presented which significantly improve the runtime of balanced buffering, as will be shown in section 4.7. Both of these techniques have been proposed to improve the balanced buffering algorithm, and have the capacity to be extended to other similar methods. One of them, *smart bound*, helps the bound to filter more non-promising solutions, while the

other one, s*olution jumper*, uses the bounding decisions on one solution to skip solving the other solutions in the same solution list.

4.6.1 Smart Bound

At the time of solving a sub-problem, the bound must be calculated to ensure a promising search strategy. From equation 3-3:

$$LowerBound = D_{current} + \alpha_b + Ideal\ Delay_{Subtree}$$

The calculated bound for a common sub-problem often varies as the sub-problem can be reached from different paths in the search space. In figure 4-23, it is shown how one common sub-problem can be met through 2 different search paths, resulting in different $D_{current}$.

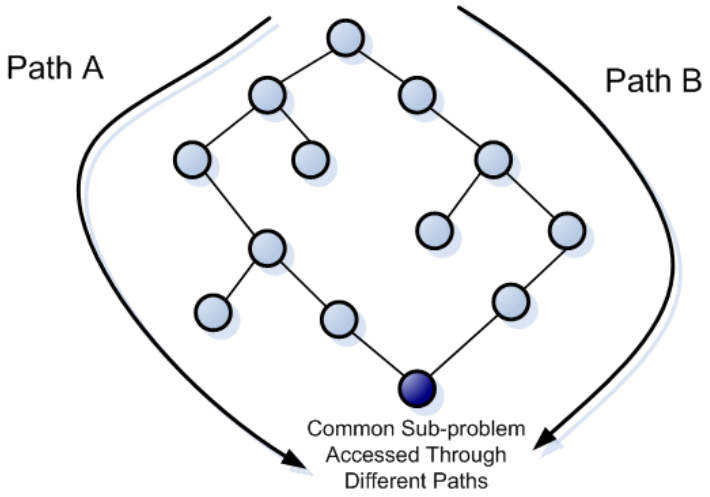


Figure 4-23 Meeting the same sub-problem through different paths

Studying the bound equation, it is observed that for common sub-problems, the only variable expression in the bound calculation is $D_{current}$, as α_b and $SubTree_{IdealDelay}$ are always constant. Now, consider the following situation:

A) A sub-problem is encountered during search space traversal and its bound is calculated as:

$$Bound_A = D_{current_A} + \alpha_b + SubTree_{IdealDelay}$$

B) The same sub-problem is re-encountered during search space traversal, but through a different search path. The bound for this case would be:

$$Bound_B = D_{current_B} + \alpha_b + SubTree_{IdealDelay}$$

If $Bound_A$ is subtracted from $Bound_B$:

$$Bound_B - Bound_A = D_{current_B} - D_{current_A}$$

Now if $D_{current_B} > D_{current_A}$, one can conclude that $Bound_B - Bound_A > 0$ or in fact:

$$Bound_B > Bound_A \qquad \text{(I)}$$

Given (I), it is clear that solving the sub-problem never leads to any better solution in case **B**, because:

1- If the sub-problem is filtered by the bound in case **A**, it will be definitely filtered in case **B**, and
2- If the sub-problem is not filtered by the bound in case **A**, its sub-subproblems are somewhere filtered by the bound. If it was not the case, it would not be needed to solve the same problem for a second time. Now given the fact that $D_{current_B}$ will be used in calculating the bound for any sub-subproblem, and also provided that $D_{current_B} > D_{current_A}$, all of the sub-subproblems will have larger values for their bounds, and so does the previously filtered sub-subproblem. As a result, solving the same sub-problem never yields any better solution in case **B**.

This property helps boosting the bound by monitoring the current delays. In fact, a sub-problem is solved only if its recent current delay is better than its previous one. This is achievable by regularly keeping a record of various current delays for a sub-problem. The implementation is very straightforward: a data field is added to each solution list to store the value of the current delay. Once a solution list is used, it is updated and used for future references.

Applying this speedup technique, many unfiltered but non-promising sub-problems are ignored while search progresses, and consequently a significant amount of runtime is saved.

4.6.2 Solution Jumper

The solutions in a typical solution list are sorted in terms of minimum delay. The idea of the solution jumper is that if a solution with better ideal delay does not manage to pass the bound, its inferior solutions cannot either. To better understand, let us look back at the bound equation:

$$LowerBound = D_{current} + \alpha_b + Ideal\ Delay_{Subtree}$$

For all the solutions in a solution list $D_{current}$ and α_b are constant, as they all share the same originating buffering problem (α_b is the intrinsic delay of the source buffer not the solution buffer). Now, if the bound is calculated for each solution, the only variable element is $Ideal\ Delay_{Subtree}$, that is the solution ideal delay. As a result, if a solution has a large ideal delay and fails the bound, one can positively conclude that all the solutions in that solution list that have larger ideal delays will be filtered by the bound. Based on this, exploring a solution list has to be stopped whenever a solution is filtered during the solving procedure.

Note that there is no need to abort the list entirely, as there might be a real solution at the end of the list, which needs to be verified. Therefore the algorithm *jumps* to the real solution at the end of the list before completely quitting the list. That is why this method is called s*olution jumper*. In figure 4-24, this procedure is shown in 3 phases. In the first phase, a solution is chosen to be solved. In the second phase, this solution is filtered by the bound and the control is returned to the solution list. Since no real delay has been obtained, the algorithm jumps to the end of the solution list where there is always a complete solution (this is because of pruning the list regularly).

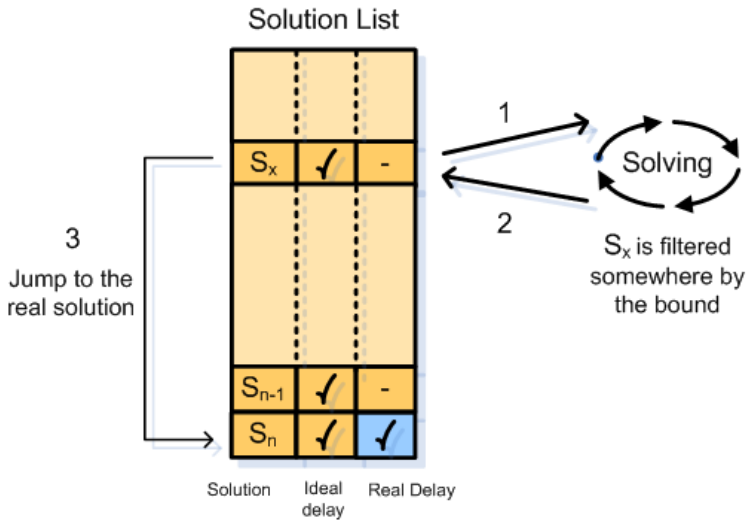


Figure 4-24 Solution jumping for a positive phase solution list

The method introduced can be easily applied to positive phase solution lists where there is a unified ordered list of solutions. However, to utilize the same idea for negative phase solution lists where there are two independent sub-lists, one has to merely jump to the end of the sub-list, instead of the end of

the solution list. This constraint makes the implementation slightly more complicated for two-part solution lists, but the overall time gained by this method is still significant, as shown in section 4.7.

4.7 Experimental Results

The same implementation conditions are used as in chapter 3. In order to make a comparison between the simple branch-and-bound method and the mixed method, the program is run for fanout numbers of 100, 200, 300, 500 and 1000, using buffer library **A** introduced in the section 3.9. The search space access method is based on LWB, whose runtime will be studied separately. The results are shown in table 4-2. The average speed up factor is 2.2 when applying the mixed method to the same set of buffering problems as in the second chapter. As shown in table 4-2, the mixed method operates with a reasonably low memory usage, while pruning the solution lists reduces memory consumption even more.

Fanout	Runtime (seconds)			Memory Usage (Kbytes)		
	Branch-and-Bound	Mixed Method		Branch-and-Bound	Mixed Method	
		Runtime	Speed up		Without Pruning	With Pruning
100	488	203	2.4	Negligible[1]	407	246
200	1936	861	2.2		644	404
300	7665	3024	2.5		1,085	702
500	14466	6483	2.2		1,428	932
1000	59077	26324	2.2		2,336	1,586

Table 4-2 Runtime and memory usage for the simple branch-and-bound and the mixed method.

The program was also run with the smart bound as well as the solution jumper methods. The speed up factors are studied in table 4-3. Note that they are compared with the simple mixed method and not with the simple branch-and-bound method. As the effect of using the mixed method on the algorithm runtime is linear, the efficiency of the smart bound and the solution jumper is examined in comparison with the simple mixed method to reduce runtime. While a slight speed up is observed with the solution jumper, the program runtime improves significantly with the smart bound, particularly for larger entries. Running the program with a combination of all speed up techniques a significant runtime reduction is observed. Furthermore, the memory usage remains reasonable even for large fanout

[1] A few kilobytes

numbers. Table 4-4 compares the runtime and the memory consumption between the simple branch-and-bound method and the system with all the speed up techniques.

Fanout	Simple Mixed Method (seconds)	Mixed Method with Smart Bound	
		Runtime (seconds)	Speed up
100	203	0.93	217
200	861	1.71	501
300	3024	3.34	905
500	6483	4.31	1503
1000	26324	9.40	2799

Fanout	Simple Mixed Method (seconds)	Mixed Method with Solution Jumper	
		Runtime (seconds)	Speed up
100	203	180	1.1
200	861	750	1.2
300	3024	2780	1.1
500	6483	5425	1.2
1000	26324	20896	1.3

Table 4-3 Runtime results for smart bound and solution jumper

Fanout	**Runtime (seconds)**			**Memory Usage (Kbytes)**	
	Branch-and-Bound	All Speed Enhancement Techniques		Branch-and-Bound	All Speed Enhancement Techniques
		Runtime	**Speed-up**		
100	488	0.20	2404	Negligible	193
200	1936	0.29	6522		260
300	7665	0.56	13615		369
500	14466	0.73	19709		425
1000	59077	1.07	54803		614
1100	> 24 hr	0.98	>87804		739
1200	> 24 hr	1.10	>77908	Negligible	841
1300	> 24 hr	0.89	>97078		770
1400	> 24 hr	1.54	>55886		963
1500	> 24 hr	1.43	>60083		882
1600	> 24 hr	1.18	>72727		893
1700	> 24 hr	1.42	>60759		998
1800	> 24 hr	1.51	>56992	Negligible	1,022
1900	> 24 hr	1.43	>60125		895
2000	> 24 hr	1.35	>63669		925
2100	> 24 hr	1.82	>47264		1,184
2200	> 24 hr	1.75	>49371		931

Table 4-4 Runtime and memory usage for simple branch-and-bound and complete system

Fanout	Runtime (seconds)			Memory Usage (Kbytes)	
	Branch-and-Bound	All Speed Enhancement Techniques		Branch-and-Bound	All Speed Enhancement Techniques
		Runtime	Speed-up		
2300	> 24 hr	1.84	>46854	Negligible	1,037
2400	> 24 hr	1.78	>48512		1,234
2500	> 24 hr	1.25	>69120		899
2600	> 24 hr	1.90	>45306		1,109
2700	> 24 hr	2.48	>34782		1,227
2800	> 24 hr	1.85	>46476		1,208
2900	> 24 hr	1.15	>74675		944
3000	> 24 hr	2.59	>33320		1,215

Table 4-5 (continue)

In order to make the mixed method efficient in terms of memory consumption, redundant solutions are pruned from the solution list. The memory footprint when performing the pruning operation on an arbitrary buffering problem is shown in figure 4-25. The balanced buffering algorithm is run on a problem with a fanout number of 300, while using buffer library **A**. The algorithm with the pruning function constructs the best buffer tree in 0.563 seconds while the effective memory usage is 369.064 Kilo-bytes. Also, the algorithm is run on the same problem but without pruning the solution list. This causes an increase of 162.748 Kilo-bytes in the memory usage.

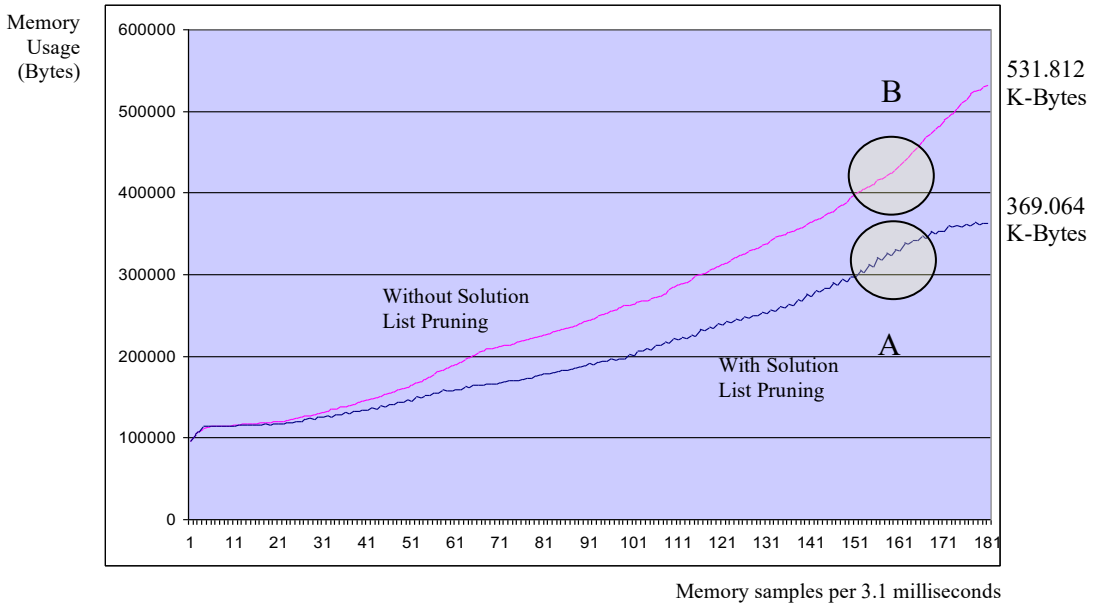


Figure 4-25 Memory tracking for a problem with a fanout number of 300

The influence of pruning the solution lists is clearly identified in two enlarged curves in figure 4-26

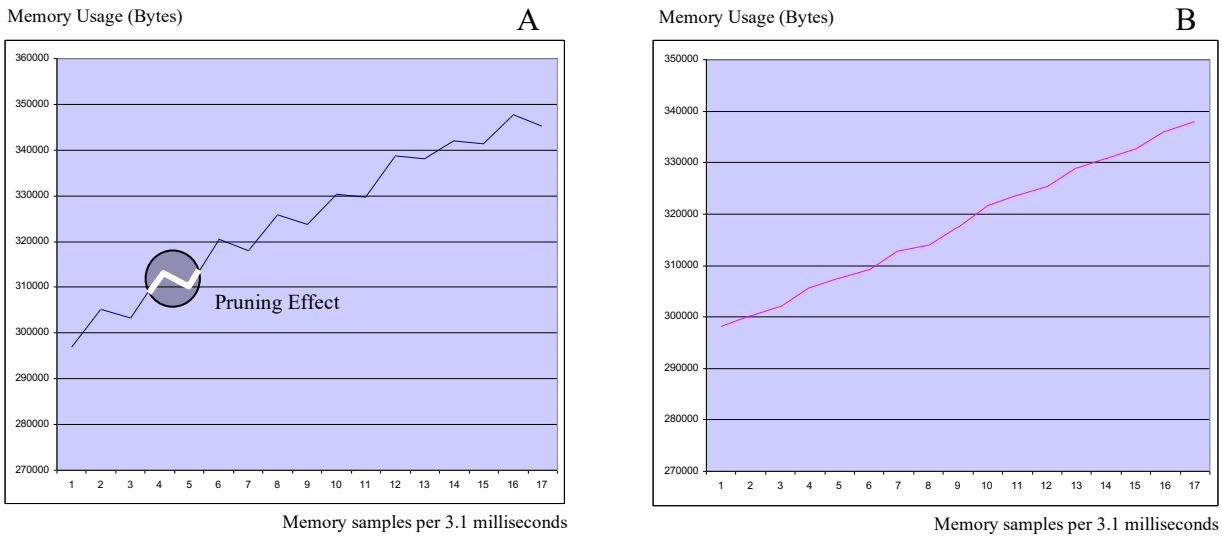


Figure 4-26 The effect of the pruning operation on memory usage.
A- The released memory is observed as a downward slope on the memory record path.
B- No pruning operation for the same set of memory samples.

In figure 4-27 the amount of released memory by pruning the solution lists of the same buffering problem is depicted. Observe that as search traversal goes on and more complete solutions become available, more redundant solutions are identified and pruned away from the solution list.

Using library **A** from section 3.9, the program is run with four different search space access methods: static-array-based, dynamic memory allocation with a simple binary search tree, dynamic memory allocation with a lazy self-reorganizing tree (LWB), and dynamic memory allocation with a sensitive self-reorganizing tree (PBB). While on average 30% speed loss is encountered by dynamic memory allocation in comparison with static memory allocation, the memory usage is reduced significantly by utilizing dynamic memory allocation, specifically for large fanout numbers. As highlighted in table 4-6, the fastest dynamic access method can be variable for different fanout numbers and as the fanout number increases the self-reorganizing operations become more efficient. Although in most of the cases the lazy self-reorganizing method operates faster than the sensitive self-reorganizing method, its performance is dominated by the sensitive self-reorganizing method for certain entries. This shows a combination of lazy and sensitive access methods would likely improve the program runtime.

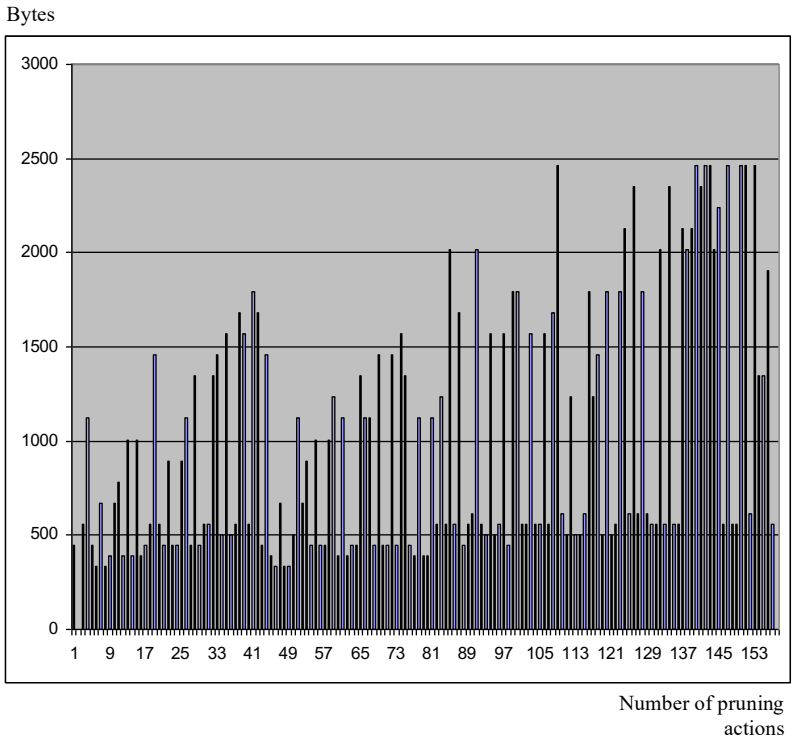


Figure 4-27 Released memory when using pruning

Fanout (*000)	Runtime (seconds)							Memory Usage (Mbytes)		
	Static Memory Allocation	Dynamic Memory Allocation						Static Memory Allocation	Dynamic Memory Allocation	
		Simple Binary Search Tree		Lazy Weight Binary Search Tree		Perfectly Balanced Binary Search Tree				
		Runtime	Delay	Runtime	Delay	Runtime	Delay		Memory	Reduction
100	11.234	14.406	+28%	14.437	+28%	14.406	+28%	18.096	5.648	69%
200	16.922	22.516	+33%	22.047	+30%	22.250	+31%	25.460	6.760	74%
300	34.735	43.079	+24%	42.860	+23%	43.797	+25%	73.184	7.516	90%
400	26.656	32.188	+24%	35.079	+31%	32.188	+24%	95.968	8.408	91%
500	22.390	29.609	+33%	30.046	+34%	29.563	+33%	71.812	9.328	87%
600	42.156	55.235	+31%	54.469	+29%	57.032	+35%	122.164	9.532	92%
700	41.500	55.656	+34%	54.796	+32%	57.875	+39%	93.216	10.780	89%
800	28.812	39.359	+37%	39.312	+37%	39.828	+37%	85.256	10.220	88%
900	93.782	126.234	+34%	121.671	+29%	126.078	+34%	123.552	10.932	91%
1000	31.093	42.375	+36%	42.359	+36%	43.297	+39%	198.216	10.460	95%

Table 4-6 Runtime and memory usage for different access methods

4.7.1 Runtime Analysis

A linear regression was previously proposed to approximately calculate the algorithm time complexity. The mixed method time complexity is estimated likewise. The trend line obtained corresponds to the linear function y = 0.05x + 0.97, with a least square value of 0.5. The trendline is demonstrated in figure 4-28. Thus, it is concluded that the mixed method affects the time complexity of the balanced buffering algorithm such that it now appears linear.

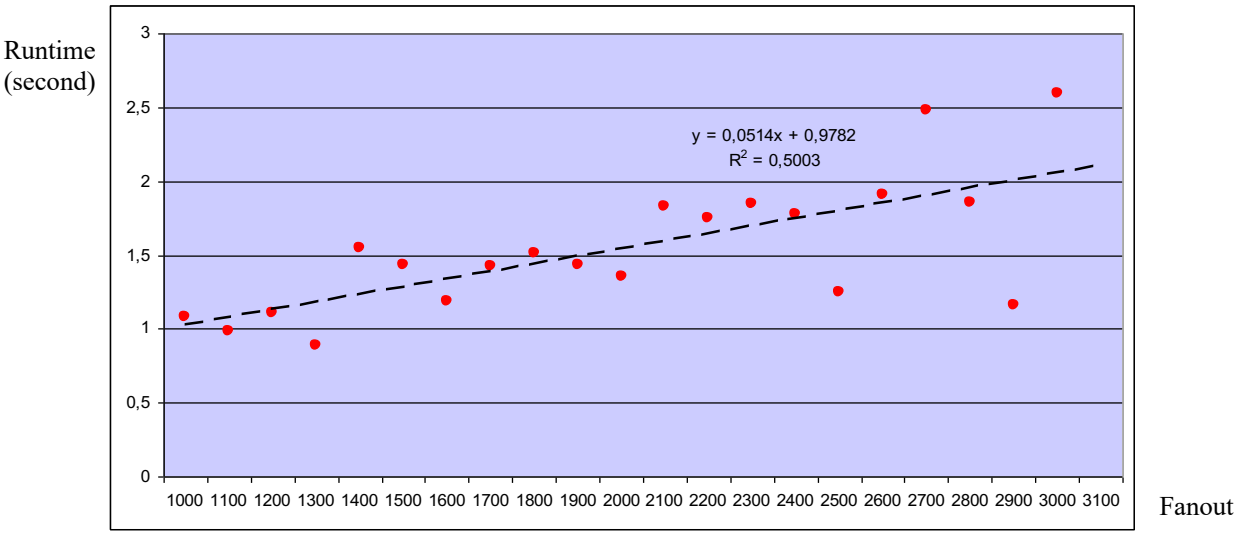


Figure 4-28 Curve-fitting on an arbitrary set of buffering problem runtime

4.8 Summary

While buffering in general is an NP-complete problem, it has been shown how one can perform balanced buffering in a reasonable amount of time and acceptable memory usage when using the proposed speed-up methods. A mixture of two problem solving methods, branch-and-bound and dynamic programming, were studied to improve the performance of the balanced buffering problem. The idea of the mixed method can be applied to solve any similar hybrid problem demonstrating both a dynamic programming structure and the possibility of having a mathematically computable bound to implement the branch-and-bound algorithm. When this occurs, one can profit from both the solution quality of the branch-and-bound method, and the fast yet memory efficient techniques of dynamic programming. On the other hand, having a reliable bound, one can also improve the dynamic programming algorithm by analytically selecting the promising sub-problems to be solved, and thus save time and memory. Also, two efficient techniques have been introduced to speed-up the search traversal: s*mart bound* and s*olution jumper*. Such speed enhancement methods are applicable to any

branch-and-bound algorithm where the feasible region contains a large number of common sub-solutions (in order to use the smart bound), and a cost-ordered solution list (in order to apply solution jumper). In addition, some useful data structures were introduced, some of which were specifically tailored to the balanced buffering problem. Interestingly, some of them, such as the two newly defined binary search trees, can have more applications in similar contexts.

CONCLUSION & FUTURE WORK

Conclusion

This thesis presents a number of methods to address the balanced buffering problem. The main contributions are listed as follows:

1- *An efficient way of balanced buffering using branch-and-bound algorithm:* While most of the buffering methods in use today are considered as extended versions of Van Ginneken's dynamic programming approach, in this thesis the buffer tree construction problem has been studied from a different perspective. A method based on the branch-and-bound technique was introduced to solve the specific case of buffering problems where loads are identical in terms of required time and input capacitance. One strong motivation to apply such exhaustive search method is to maximize the solution quality for the produced buffer tree, which is generally achieved due to the high number of possible solutions examined during the search traversal. As the nature of a typical balanced buffering problem demonstrates symmetry of structure in various aspects, a recursive structure was identified and applied to implement the branch-and-bound algorithm. One achievement was also done when the timing properties of an ideal buffer tree were mathematically characterized. The ideal delay equation for the general case of balanced buffering was successfully extracted and applied to design a filtering lower bound for the branch-and-bound method. While the size of the search space grows up exponentially with the increase in the fanout number, the bound introduced can efficiently cut large portions of the search space that contain no solution. This was shown in section 3.9. The proposed method also takes a number of key design factors into account including buffer library, buffer tree topology and phase shifting in the presence of inverting buffers.

2- *Speed up techniques for the proposed balanced buffering algorithm:* In order to improve the runtime of the balanced buffering method, 3 speed-up techniques were proposed:

First, it was shown how one can take advantage of the existence of common sub-solutions in the search space to prevent redundant calculations. To that effect, the memory-reuse concepts of dynamic programming were applied to save the sub-problems results and reused them for

quickly solving similar sub-problems. This method was called the mixed branch-and-bound and dynamic programming method, or simply the mixed method. As opposed to the simple branch-and-bound method where no solution list is kept, the mixed method needs to save the information of those solution lists, and therefore, the memory consumption issue becomes crucial. A solution list pruning technique was presented to effectively reduce memory consumption by frequently relaxing the solution lists of non-promising solutions. Pruning technique resulted in implementing the mixed method with a reasonable memory foot-print.

Secondly, the smart bound method was proposed to improve the buffering algorithm runtime by keeping a record of filtered sub-solutions in the search space. That technique led to a significant speed-up according to the test results provided in section 4.7.

Finally, the solution jumper method was introduced to prevent exploring non-promising sub-problems by ignoring the solutions dominated by a filtered solution in the solution list. The combination of all these techniques, together with the pruning method, results in a significantly fast buffering algorithm, while the memory consumption remains reasonable.

3- *Self-reorganizing binary search trees:* Due to the need of saving and accessing the solutions lists by the mixed method, two new classes of dynamic binary search trees were presented in this thesis: LWB and PBB. The proposed data structures are able to modify their structure such that highly referenced solution lists can be accessed faster than the other solution lists. This is achieved by keeping a record of the usage number of each solution list and using it in ranking and re-organizing the tree-nodes. While LWB is less sensitive than PBB, both of them have shown to be more effective than a simple binary search tree in reducing the balanced buffering runtime.

Future Work

The potential extensions of the work presented in this thesis are briefly reviewed, which together can help generalizing balanced buffering method to solve more general types of buffering problems.

As devices shrink in size, deep submicron designs demonstrate the increasing importance of interconnect delay on the circuit performance. As a result many research groups are at work to tackle the issue of interconnect delay, many of which have broadened their focus to also study new materials employed in the circuit fabrication process or even new approaches to help signals traveling faster

through a conductor. For instance, new studies at Caltech and Stanford University argue the possibility of achieving faster integrated circuits by combining the advantages of photonics and electronics [OZMAY, 2006]. The new method, called plasmonic, would help chipmaker design circuit components with frequencies 100,000 times greater than the ones of current microprocessor, with almost no wire capacitance. Nevertheless, it is still required to develop and improve buffer insertion techniques in order to efficiently handle the challenging issue of interconnect delay, as such innovative techniques are yet to be industrialized. As a result, one important extension of balanced buffering is consider the wire's physical characteristics.

It has been assumed that all loads have identical required time and input capacitance. One important extension of balanced buffering is to construct a buffer tree for dissimilar sinks by using the ideas of balanced buffering. To that effect, it is necessary to mathematically find a way to transform the dissimilar sinks into identical ones in order to profit from the efficient balanced buffering algorithm. Yet, the problem should not be over-simplified in order to preserve the solution quality.

As discussed in section 4.5.7, there exists a possibility of achieving even faster search space access times by combining LWB and PBB. The success of combining those methods is dependent on finding a proper **m**, the number of adequate solution list references before altering the access method from a lazy self-reorganizing tree to a sensitive one. The likelihood of the existence of that value could be an interesting subject for further studies.

Some other subjects in this domain include higher order delay models, better signal characterizations by considering different rise/fall times, buffer sizing, and resource reduction for non-critical paths in the buffer tree.

APPENDICES

APPENDIX A

Minimum delay calculation for the balanced buffer tree

1. The Minimum Delay Equation

Theorem: given the source output resistance β_0, and the total load capacitance γ_T, the optimum delay is obtained from the following equation:

$$D_{optimum} = \mu\left(1 + \ln\left(\frac{\beta_0 \gamma_T}{\mu}\right)\right) \qquad \text{A-1}$$

where μ is:

$$\mu = \beta_b \gamma_b e^{\left(\frac{\alpha_b}{\mu}+1\right)} \qquad \text{A-2}$$

and β_b and γ_b are respectively the output resistance and the input capacitance of the best buffer in the library and α_b represents its intrinsic delay. As is shown in appendix C, the best buffer is the one with the smallest μ in a given buffer library.

It is necessary to initially find a well-formed delay equation for a balanced buffer tree, and then examine under which conditions the minimum delay value is obtained. Let us begin with calculating the delay equation for cases having zero, one and two levels of buffering. Next, the delay equation for K levels of buffering will be found and the optimum delay will be calculated with respect to K.

2. Proof

Pre-assumptions for minimum delay calculation

1) Continuous rather than discrete values are allowed for the number of levels and branches in the ideal buffer tree.

2) Due to the symmetrical structure of a balanced buffer tree, all source-to-sink paths can be considered as the critical path.

Delay with zero levels of buffers

Having a source output resistance of β_0 and a total load capacitance of γ_T like what is shown in figure A-1 the delay equation of a simple tree with no buffer inserted is:

$$D = \beta_0 \gamma_T \qquad \text{A-3}$$

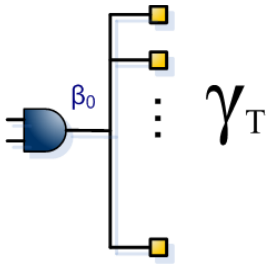


Figure A-1 Zero level buffering

Delay with one level of buffers

Regarding the buffer tree structure in figure A-2 and using a certain buffer type B_1, the delay equation for the buffer tree would be:

$$D = \beta_0 n_1 \gamma_1 + \frac{\beta_1 \gamma_T}{n_1} + \alpha_1 \qquad \text{A-4}$$

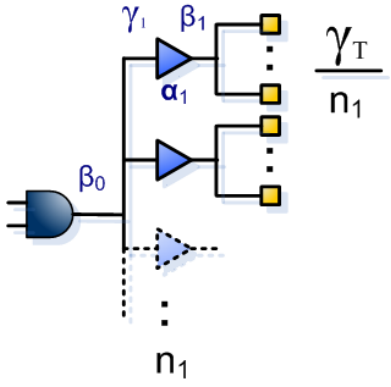


Figure A-2 One level buffering

Where β_0 is the source output resistance, γ_T is the total load capacitance, β_1, γ_1 and α_1 are respectively the output resistance, the input capacitance and the intrinsic delay of the buffer B_1, and n_1 is the number of buffers used in the tree. In order to find the minimum delay, the derivative of D is taken with respect to n_1:

$$\frac{\partial D}{\partial n_1} = \beta_0 \gamma_1 - \frac{\beta_1 \gamma_T}{n_1^2} \equiv 0 \qquad \text{A-5}$$

Solving the above equation yields the best number of buffers:

$$n_{1_{optimum}} = \sqrt{\frac{\beta_1 \gamma_T}{\beta_0 \gamma_1}} \qquad \text{A-6}$$

The optimum delay is found by replacing the best number of buffers in the original delay equation:

$$D_{optimum} = \alpha_1 + 2\sqrt{\beta_0 \beta_1 \gamma_1 \gamma_T} \qquad \text{A-7}$$

Delay with two levels of buffers

The delay of a buffer tree with 2 levels of buffering is now calculated, as shown in figure A-3. To preserve generality, it is assumed that different buffers are used at each level, say B_1 and B_2. The same notation are used as in equation A-7 to show source output resistance and total load capacitance, while the physical properties of each applied buffer are symbolized as β_i, γ_i and α_i for each buffer. n_1 and n_2 are the number of buffers for the first and the second level respectively. The delay is calculated as:

$$D = \beta_0 n_1 \gamma_1 + \alpha_1 + \beta_1 n_2 \gamma_2 + \alpha_2 + \frac{\beta_2 \gamma_T}{n_1 n_2} \qquad \text{A-8}$$

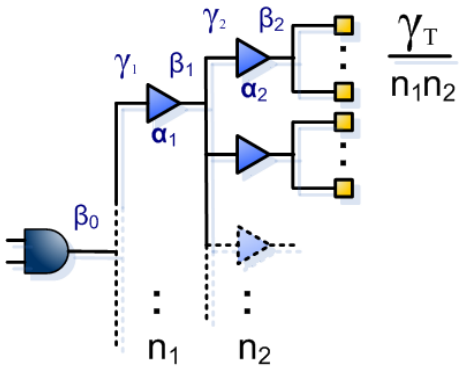


Figure A-3 Two level buffering

As there are now 2 variables in the delay equation, n_1 and n_2, in order to find the optimum delay partial derivative should be taken, once with respect to n_1 and once with respect to n_2. By setting the two resulting equations, the best buffer numbers are then found for each buffering level.

$$\frac{\partial D}{\partial n_1} = \beta_0\gamma_1 - \frac{\beta_2\gamma_T}{n_1^{\,2} n_2} \equiv 0 \qquad \text{A-9}$$

$$\frac{\partial D}{\partial n_2} = \beta_1\gamma_2 - \frac{\beta_2\gamma_T}{n_1 n_2^{\,2}} \equiv 0 \qquad \text{A-10}$$

From (A-9): $$\frac{1}{n_2^{\,2}} = \frac{n_1^{\,4}\beta_0^{\,2}\gamma_1^{\,2}}{\beta_2^{\,2}\gamma_T^{\,2}} \qquad \text{A-11}$$

Into (A-10): $$\beta_1\gamma_2 = \frac{\beta_2\gamma_T}{n_1}.\frac{n_1^{\,4}\beta_0^{\,2}\gamma_1^{\,2}}{\beta_2^{\,2}\gamma_T^{\,2}} \Rightarrow n_1^{\,3} = \frac{\beta_0\beta_2\gamma_1\gamma_T}{\beta_0^{\,2}\gamma_1^{\,2}} \qquad \text{A-12}$$

Similarly, by putting (A-10) into (A-9):

$$n_2^{\,3} = \frac{\beta_0\beta_2\gamma_1\gamma_T}{\beta_1^{\,2}\gamma_2^{\,2}} \qquad \text{A-13}$$

Replacing (A-12) and (A-13) into (A-8), the optimum delay for a buffer tree with 2 levels of buffers is:

$$D_{optimum} = \alpha_1 + \alpha_2 + 3\sqrt[3]{\beta_0\beta_1\beta_2\gamma_1\gamma_2\gamma_T} \qquad \text{A-14}$$

Delay with *K* levels of buffers

Using inductive reasoning, a general delay equation for K levels of buffers with dissimilar buffer types is found with the properties shown in figure A-4.

$$D = \sum_{i=0}^{K} \alpha_i + (K+1)\left(\beta_0 \gamma_T \cdot \beta_1 \ldots \beta_K \cdot \gamma_1 \ldots \gamma_K\right)^{\frac{1}{K+1}} \qquad \text{A-15}$$

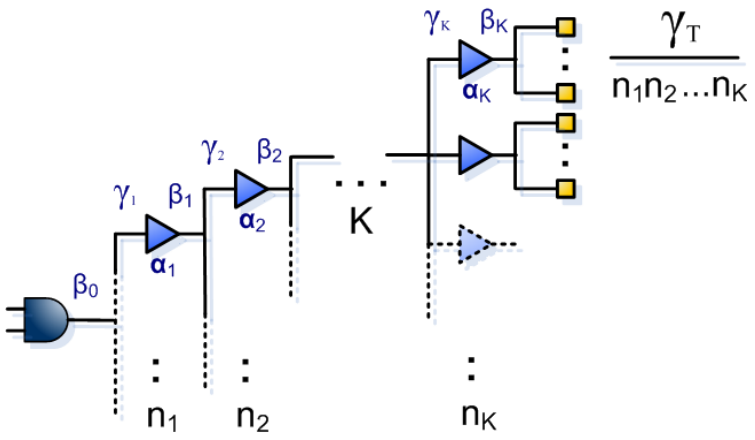


Figure A-4 K level buffering

The smallest D must be found. Therefore, a buffer in the library should be used for which the product of β_i and γ_i is minimum. Let us call this buffer *best buffer*. It is proven in appendix B that replacing all buffers with this best buffer results in a delay less than the delay value obtained from (A-15). By doing so, the delay equation becomes:

$$D = K\alpha_b + (K+1)\left(\beta_0 \gamma_T \left(\beta_b \gamma_b\right)^K\right)^{\frac{1}{K+1}} \qquad \text{A-16}$$

The smallest value for D will occur for some number of levels K. For that number of levels K, the function will be at a minimum. Therefore, the derivative of D is taken with respect to K in order to find the best number of levels.

$$\frac{\partial D}{\partial K} \equiv 0 \qquad \text{A-17}$$

Let us use a new variable μ, where:

$$\mu = \left(\beta_0 \gamma_T \left(\beta_b \gamma_b \right)^K \right)^{\frac{1}{K+1}} \quad \text{A-18}$$

It then gives:

$$D = \mu + (\mu + \alpha_b) K \quad \text{A-19}$$

Knowing that:

$$\frac{\partial D}{\partial K} = \frac{\partial D}{\partial \mu} . \frac{\partial \mu}{\partial K} \equiv 0 \quad \text{A-20}$$

From (A-16):

$$\frac{\partial D}{\partial K} = \mu + \alpha_b + (K+1) . \frac{\partial \mu}{\partial K} \quad \text{A-21}$$

In order to remove K from (A-21), (A-18) is manipulated and plugged in (A-21). This is done in the following manner:

$$\mu = \left(\beta_0 \gamma_T \left(\beta_b \gamma_b \right)^K \right)^{\frac{1}{K+1}}$$

$$\Rightarrow \quad \ln \mu = \frac{1}{(K+1)} \left[\ln (\beta_0 \gamma_T) + K . \ln (\beta_b \gamma_b) \right]$$

$$\Rightarrow \quad (K+1) . \ln \mu = \ln (\beta_0 \gamma_T) + K . \ln (\beta_b \gamma_b)$$

$$\Rightarrow \quad K \left(\ln \mu - \ln (\beta_b \gamma_b) \right) = \ln (\beta_0 \gamma_T) - \ln \mu$$

$$\Rightarrow \quad dK \left(\ln \mu - \ln (\beta_b \gamma_b) \right) + K \frac{d\mu}{\mu} = -\frac{d\mu}{\mu}$$

$$\Rightarrow \quad d\mu \frac{(K+1)}{\mu} = dK \left(\ln (\beta_b \gamma_b) - \ln \mu \right)$$

$$\Rightarrow \quad \frac{d\mu}{dK} = \frac{\mu}{(K+1)} \ln \left(\frac{\beta_b \gamma_b}{\mu} \right) \quad \text{A-22}$$

Plugging (A-22) into (A-21):

$$\frac{\partial D}{\partial K} = \mu + \alpha_b + \mu . \ln \left(\frac{\beta_b \gamma_b}{\mu} \right) \equiv 0 \quad \text{A-23}$$

Now μ corresponding to the minimum delay can be found:

$$\frac{\partial D}{\partial K} = \mu + \alpha_b + \mu.\ln\left(\frac{\beta_b \gamma_b}{\mu}\right) \equiv 0$$

$$\Rightarrow \quad \mu.\left(\ln\left(\frac{\beta_b \gamma_b}{\mu}\right) + 1\right) = -\alpha_b$$

$$\Rightarrow \quad \ln\left(\frac{\beta_b \gamma_b}{\mu}\right) = -\frac{\alpha_b}{\mu} - 1$$

$$\Rightarrow \quad \ln \beta_b \gamma_b - \ln \mu = -\frac{\alpha_b}{\mu} - 1$$

$$\Rightarrow \quad \ln \mu = +\frac{\alpha_b}{\mu} + 1 + \ln \beta_b \gamma_b$$

$$\Rightarrow \quad \mu = \beta_b \gamma_b e^{\left(\frac{\alpha_b}{\mu}+1\right)} \qquad \text{A-24}$$

$$\Rightarrow \quad \ln\left(\frac{\mu}{\beta_b \gamma_b}\right) = \frac{\alpha_b}{\mu} + 1 \qquad \text{A-25}$$

Using (A-18), the optimum number of levels, K is found and will be later used to find the minimum delay:

$$\mu = \left(\beta_0 \gamma_T \left(\beta_b \gamma_b\right)^K\right)^{\frac{1}{K+1}}$$

$$\Rightarrow \quad \ln \mu = \frac{1}{(K+1)}\left[\ln\left(\beta_0 \gamma_T\right) + K.\ln\left(\beta_b \gamma_b\right)\right]$$

$$\Rightarrow \quad (K+1).\ln \mu - K.\ln\left(\beta_b \gamma_b\right) = \ln\left(\beta_0 \gamma_T\right)$$

$$\Rightarrow \quad K.\left(\ln \mu - \ln\left(\beta_b \gamma_b\right)\right) = \ln\left(\beta_0 \gamma_T\right) - \ln \mu$$

$$\Rightarrow \quad K = \frac{\ln\left(\frac{\beta_0 \gamma_T}{\mu}\right)}{\ln\left(\frac{\mu}{\beta_b \gamma_b}\right)} \qquad \text{A-26}$$

From (A-19) the optimum delay is now calculated:

$$D = \mu + (\mu + \alpha_b).K$$

Using (A-26):

$$D = \mu + (\mu + \alpha_b).\frac{\ln\left(\frac{\beta_0\gamma_T}{\mu}\right)}{\ln\left(\frac{\mu}{\beta_b\gamma_b}\right)} \quad \text{A-27}$$

Plugging (A-14) into (A-27):

$$D_{optimum} = \mu + (\mu + \alpha_b).\frac{\ln\left(\frac{\beta_0\gamma_T}{\mu}\right)}{\frac{(\mu + \alpha_b)}{\mu}} \quad \text{A-28}$$

$$D_{optimum} = \mu + \mu.\ln\left(\frac{\beta_0\gamma_T}{\mu}\right) \quad \text{A-29}$$

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

Where

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

These are equations A-1 and A-2, which appeared in the theorem.

■

Corollary. The best number of levels ($K_{optimum}$) is calculated using the optimum delay equation.

From (A-16):

$$D_{optimum} = \mu + (\mu + \alpha_b).K_{optimum}$$

And from (A-1):

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

$$K_{optimum} = \frac{\mu}{\mu + \alpha_b}\ln\left(\frac{\beta_0\gamma_T}{\mu}\right) \quad \text{A-29}$$

$$K_{optimum} = \frac{1}{1 + \frac{\alpha_b}{\mu}}\ln\left(\frac{\beta_0\gamma_T}{\mu}\right) \quad \text{A-30}$$

APPENDIX B

Proof of the Best Buffer Existence

Theorem: The ideal buffer tree only contains one type of buffer.

It is first assumed that there are at least two different buffer types in the ideal buffer tree. It will then be shown that this assumption does not lead to any better delay than if only a single buffer type would be used. To this end, the optimum delay of such a buffer tree has to be found. Since it is difficult to find a closed form formula for the delay function being extremized, the *Lagrange multipliers*[1] method is used. The *L*agrange multipliers method is very useful to find the extrema (maxima or minima) of a function subject to a fixed constraint. This method is used to find the optimum delay for a buffer tree made of two buffer types. In order to apply the Lagrange multipliers method, a constraint has to be defined first. Assuming that m_1 levels of the ideal buffer tree contain buffer type B_1, and the rest, say m_2, contain buffer type B_2, the total number of buffering levels, k, would then be:

$$k = m_1 + m_2$$

Having this, a very simple constraint can be defined as:

$$G = m_1 + m_2 - k$$

The optimum delay of a buffer tree with 2 buffer types is now calculated subject to the above condition. For more convenience and clarity, in figure B-1 the necessary steps to find the values for m_1 and m_2 that yield the optimum delay are summarized. After finding minimum m_1 and minimum m_2, it will be discussed whether it is impossible to have a faster buffer tree by multi-type buffering.

[1]The Lagrange Multipliers method is named after Joseph Louis Lagrange, an Italian mathematician and astronomer who made important contributions to all fields of analysis and number theory.

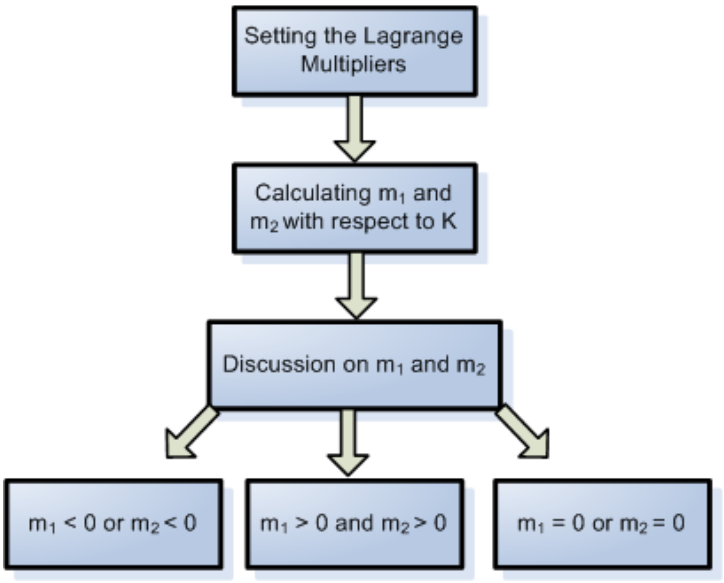


Figure B-1 The proof procedure using Lagrange Multipliers

Setting the Lagrange Multipliers

For simplicity, β and γ are merged into γ without loss of generality. Using the general delay equation (B-1) [1], the specific case of k levels of buffers and 2 different buffer types is obtained (B-2):

$$D = \sum_{i=0}^{k} \alpha_i + (k+1)\left(\beta_0 \gamma_T . \beta_1 ... \beta_k . \gamma_1 ... \gamma_k\right)^{\frac{1}{k+1}} \quad \text{B-1}$$

$$D = m_1\alpha_1 + m_2\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} \quad . \quad \text{B-2 (delay equation for two buffer types)}$$

As introduced before, the necessary constraint for the Lagrange multiplier is:

$$G = m_1 + m_2 - k \quad \text{B-3}$$

The main function being minimized is defined in terms of m_1, m_2 and k, introducing a new unknown, λ, as follows:

$$\Phi\left(m_1, m_2, k, \lambda\right) = D - \lambda G \quad \text{B-4}$$

The overall procedure to solve such problems is to first find λ, and once λ is found m_1 and m_2 are computed. In this case, since it is sufficient to study the necessary conditions for m_1 and m_2

[1] Equation A-15 from appendix A

under which the minimum delay is found, calculations will be stopped after obtaining the minimum m_1 and m_2. The possibility of having an optimum delay with 2 buffer types will be discussed. From (B-2), (B-3) and (B-4):

$$\Phi = m_1\alpha_1 + m_2\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} - \lambda\left(m_1 + m_2 - k\right)$$

The smallest Φ subject to the constraint is needed. Therefore, derivatives with respect to m_1 and m_2 are separately taken:

$$\left(\frac{\partial\Phi}{\partial m_1}\right)_{m_2,k,\lambda:const} \equiv 0$$

$$\alpha_1 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} . \frac{Log_e\left(\gamma_1\right)}{\left(k+1\right)} - \lambda = 0 \qquad \text{B-5}$$

Similarly an expression is found for m_2:

$$\left(\frac{\partial\Phi}{\partial m_2}\right)_{m_1,k,\lambda:const} \equiv 0$$

$$\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} . \frac{Log_e\left(\gamma_2\right)}{\left(k+1\right)} - \lambda = 0 \qquad \text{B-6}$$

Preparing an auxiliary equation

From (B-5) and (B-6):

$$\alpha_1 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} . \frac{Log_e\left(\gamma_1\right)}{\left(k+1\right)} = \alpha_2 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} . \frac{Log_e\left(\gamma_2\right)}{\left(k+1\right)}$$

$$\frac{\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}}}{\left(k+1\right)} . Log_e\left(\frac{\gamma_1}{\gamma_2}\right) = \left(\alpha_2 - \alpha_1\right) \qquad \text{B-7}$$

$$\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} = \frac{\left(k+1\right).\left(\alpha_2 - \alpha_1\right)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}$$

$$\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right] = \left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)^{(k+1)} \qquad \text{B-8}$$

Calculating m_1 with respect to k

From $k = m_1 + m_2$:

$$m_2 = k - m_1$$

Into (B-7):

$$\frac{\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{(k-m_1)}\right]^{\frac{1}{k+1}}}{(k+1)}.Log_e\left(\frac{\gamma_1}{\gamma_2}\right) = (\alpha_2-\alpha_1)$$

Rearranging with respect to m_1:

$$\left(\gamma_T {\gamma_2}^k\right)^{\frac{1}{k+1}}.\left(\frac{\gamma_1}{\gamma_2}\right)^{\frac{m_1}{k+1}} = \frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}$$

$$\left(\frac{\gamma_1}{\gamma_2}\right)^{\frac{m_1}{k+1}} = \frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}.\frac{1}{\left(\gamma_T {\gamma_2}^k\right)^{\frac{1}{k+1}}}$$

$$\left(\frac{\gamma_1}{\gamma_2}\right)^{m_1} = \left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)^{(k+1)}.\frac{1}{\left(\gamma_T {\gamma_2}^k\right)}$$

$$m_1 Log_e\left(\frac{\gamma_1}{\gamma_2}\right) = (k+1).\left(Log_e\left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) - Log_e\left(\gamma_T {\gamma_2}^k\right)$$

$$m_1 = \frac{(k+1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}.\left(Log_e\left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) - \frac{Log_e\left(\gamma_T {\gamma_2}^k\right)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)} \qquad \text{B-9}$$

Calculating m_2 with respect to k

From $k = m_1 + m_2$:

$$m_1 = k - m_2$$

Into (B-7):

$$\frac{\left[\gamma_T \gamma_1^{(k-m_2)} \gamma_2^{m_2}\right]^{\frac{1}{k+1}}}{(k+1)} . Log_e\left(\frac{\gamma_1}{\gamma_2}\right) = (\alpha_2 - \alpha_1)$$

Rearranging with respect to m_2:

$$\left(\gamma_T \gamma_1^{k}\right)^{\frac{1}{k+1}} \cdot \left(\frac{\gamma_2}{\gamma_1}\right)^{\frac{m_2}{k+1}} = \frac{(k+1).(\alpha_2 - \alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}$$

$$\left(\frac{\gamma_2}{\gamma_1}\right)^{\frac{m_2}{k+1}} = \frac{(k+1).(\alpha_2 - \alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)} \cdot \frac{1}{\left(\gamma_T \gamma_1^{k}\right)^{\frac{1}{k+1}}}$$

$$\left(\frac{\gamma_2}{\gamma_1}\right)^{m_2} = \left(\frac{(k+1).(\alpha_2 - \alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)^{(k+1)} \cdot \frac{1}{\left(\gamma_T \gamma_1^{k}\right)}$$

$$m_2 Log_e\left(\frac{\gamma_2}{\gamma_1}\right) = (k+1).\left(Log_e\left(\frac{(k+1).(\alpha_2 - \alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) - Log_e\left(\gamma_T \gamma_1^{k}\right)$$

$$m_2 = \frac{(k+1)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} . \left(Log_e\left(\frac{(k+1).(\alpha_2 - \alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) - \frac{Log_e\left(\gamma_T \gamma_1^{k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} \qquad \text{B-10}$$

Rewriting m_1 and m_2

Since it is known that:

$$Log_e\left(\frac{\gamma_1}{\gamma_2}\right) = -Log_e\left(\frac{\gamma_2}{\gamma_1}\right)$$

From (B-7):

$$m_1 = -\frac{(k+1)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}.\left(Log_e\left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) - \frac{Log_e\left(\gamma_T{\gamma_2}^k\right)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}$$

To simplify the equations obtained so far, A is defined as:

$$A = \frac{(k+1)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}.\left(Log_e\left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)\right) \qquad \text{B-11}$$

Also, the equivalent of (B-11) for future references is found. From (B-8):

$$\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right] = \left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)^{(k+1)}$$

Taking natural logarithm from both sides:

$$Log_e\frac{\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]}{(k+1)} = Log_e\left(\frac{(k+1).(\alpha_2-\alpha_1)}{Log_e\left(\frac{\gamma_1}{\gamma_2}\right)}\right)$$

From (B-11):

$$A = \frac{(k+1)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}\frac{Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]}{(K+1)}$$

An equivalent for A is calculated:

$$A = \frac{Log_e\left[\gamma_T \gamma_1^{m_1} \gamma_2^{m_2}\right]}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} \quad \text{B-12}$$

Rewriting m_1 and m_2 in terms of A:

$$m_1 = -A + \frac{Log_e\left(\gamma_T \gamma_2^{k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} \quad \text{B-13}$$

$$m_2 = A - \frac{Log_e\left(\gamma_T \gamma_1^{k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} \quad \text{B-14}$$

Discussion on m_1 and m_2

It is of interest to know if there exist any permissible values for m_1 and m_2 with which one can obtain an optimum delay. Therefore, different values of m_1 and m_2 are investigated which are, as mentioned before, the number of buffer levels containing buffers of type 1 and 2 respectively. All the possible values for m_1 and m_2 are summarized in three cases:

Case 1: $m_1 < 0$ OR $m_2 < 0$

Since negative values for the number of levels are not feasible, this case is not considered as a possible range for finding an optimum delay.

Case 2: $m_1 = 0$ OR $m_2 = 0$

It is clear that an answer to $k = m_1 + m_2$ would be when m_1 or m_2 equals to zero and the other one equals to k. In this case, the original delay equation

$$D = m_1\alpha_1 + m_2\alpha_2 + \left[\gamma_T \gamma_1^{m_1} \gamma_2^{m_2}\right]^{\frac{1}{k+1}} \quad \text{B-15}$$

can be rewritten with only one buffer type, i.e.:

$$D = k.\alpha_1 + \left[\gamma_T \gamma_1^{k}\right]^{\frac{1}{k+1}}$$

Therefore, for the case of having either m_1 or m_2 equal to zero, the claim is proved.

Case 3: $m_1 > 0$ AND $m_2 > 0$

The conditions under which k, m_1 and m_2 are all positive must be examined, if there are any. Initially it is assumed that $\gamma_2 > \gamma_1$ and mirror reasoning is used for the case of $\gamma_2 < \gamma_1$. The condition in which $\gamma_1 = \gamma_2$ is not studied, as according to some equations like (B-8) it is forbidden to have $\gamma_1 = \gamma_2$ (because of the log statement in the denominator). Seen from a different point of view, having $\gamma_1 = \gamma_2$ means the two types of buffers (type 1 and 2) are identical.

In order to have positive m_1 and m_2, two conditions must be shown:

I) For m_1:

$$0 < m_1 < k$$

From (B-11):

$$0 < -A + \frac{Log_e\left(\gamma_T \gamma_2^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} < k$$

$$0 < -\left(A - \frac{Log_e\left(\gamma_T \gamma_2^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}\right) < k$$

$$-k < A - \frac{Log_e\left(\gamma_T \gamma_2^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} < 0 \qquad \text{B-16}$$

II) And for m_2:

$$0 < m_2 < k$$

From (B-14):

$$0 < A - \frac{Log_e\left(\gamma_T \gamma_1^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} < k \qquad \text{B-17}$$

Merging (B-16) and (B-17) yields:

$$-k < A - \frac{Log_e\left(\gamma_T \gamma_2^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} < A - \frac{Log_e\left(\gamma_T \gamma_1^{\,k}\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)} < k$$

$$-k-A<-\frac{Log_e\left(\gamma_T{\gamma_2}^k\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}<-\frac{Log_e\left(\gamma_T{\gamma_1}^k\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}<k-A$$

$$A+k>\frac{Log_e\left(\gamma_T{\gamma_2}^k\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}>\frac{Log_e\left(\gamma_T{\gamma_1}^k\right)}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}>A-k \qquad \text{B-18}$$

Since it has been assumed that $\gamma_2>\gamma_1$, both sides of the inequality can be multiplied by $Log_e\left(\frac{\gamma_2}{\gamma_1}\right)$ without any sign change:

$$(A+k).Log_e\left(\frac{\gamma_2}{\gamma_1}\right)>Log_e\left(\gamma_T{\gamma_2}^k\right)>Log_e\left(\gamma_T{\gamma_1}^k\right)>(A-k).Log_e\left(\frac{\gamma_2}{\gamma_1}\right)$$

$$A.Log_e\left(\frac{\gamma_2}{\gamma_1}\right)+k.Log_e\left(\frac{\gamma_2}{\gamma_1}\right)>Log_e\left(\gamma_T{\gamma_2}^k\right)>Log_e\left(\gamma_T{\gamma_1}^k\right)>A.Log_e\left(\frac{\gamma_2}{\gamma_1}\right)-k.Log_e\left(\frac{\gamma_2}{\gamma_1}\right),$$

B-19

From (B-12):

$$A=\frac{Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]}{Log_e\left(\frac{\gamma_2}{\gamma_1}\right)}$$

Into (B-19):

$$Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]+k.Log_e\left(\frac{\gamma_2}{\gamma_1}\right)>Log_e\left(\gamma_T{\gamma_2}^k\right)>Log_e\left(\gamma_T{\gamma_1}^k\right)>Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]-k.Log_e\left(\frac{\gamma_2}{\gamma_1}\right)$$

Rearranging the left most and the right most expressions:

$$Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]+Log_e\left(\frac{\gamma_2}{\gamma_1}\right)^k>Log_e\left(\gamma_T{\gamma_2}^k\right)>Log_e\left(\gamma_T{\gamma_1}^k\right)>Log_e\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]-Log_e\left(\frac{\gamma_2}{\gamma_1}\right)^k$$

$$Log_e\left(\left(\frac{\gamma_2}{\gamma_1}\right)^k.\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]\right)>Log_e\left(\gamma_T{\gamma_2}^k\right)>Log_e\left(\gamma_T{\gamma_1}^k\right)>Log_e\left(\left(\frac{\gamma_1}{\gamma_2}\right)^k.\left[\gamma_T{\gamma_1}^{m_1}{\gamma_2}^{m_2}\right]\right)$$

The inequality is monotonic, so it is possible to drop the natural logarithm from all expressions:

$$\left(\frac{\gamma_2}{\gamma_1}\right)^k \cdot\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right] > \gamma_T {\gamma_2}^k > \gamma_T {\gamma_1}^k > \left(\frac{\gamma_1}{\gamma_2}\right)^k \cdot\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right] \qquad \text{B-20}$$

From the most left inequality of (B-20):

$$\left(\frac{\gamma_2}{\gamma_1}\right)^k \cdot\left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right] > \gamma_T {\gamma_2}^k$$

Rearranging:

$$\left(\frac{\gamma_2}{\gamma_1}\right)^k \cdot {\gamma_1}^{m_1} {\gamma_2}^{m_2} > {\gamma_2}^k \quad \Rightarrow \quad {\gamma_1}^k < {\gamma_1}^{m_1} {\gamma_2}^{m_2} \qquad \text{(II) B-21}$$

Therefore, as long as $\gamma_1 < \gamma_2$, ${\gamma_1}^{m_1} {\gamma_2}^{m_2}$ can be replaced in the original delay equation with ${\gamma_1}^k$ to have smaller delay value. This means the delay equation would become:

$$D = m_1\alpha_1 + m_2\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1} {\gamma_2}^{m_2}\right]^{\frac{1}{k+1}} \quad \Rightarrow \quad D = m_1\alpha_1 + m_2\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1+m_2} {\gamma_2}^{0}\right]^{\frac{1}{k+1}}$$

Since the share of the second buffer is zero in the parenthesis, and hence in the buffer tree, it has no share in the sum part $m_1\alpha_1 + m_2\alpha_2$ either:

$$D = (m_1 + m_2)\alpha_1 + 0.\alpha_2 + \left[\gamma_T {\gamma_1}^{m_1+m_2} {\gamma_2}^{0}\right]^{\frac{1}{k+1}}$$

Therefore:

$$D = k.\alpha_1 + \left[\gamma_T {\gamma_1}^{k}\right]^{\frac{1}{k+1}}$$

where $k = m_1 + m_2$. D is minimum as long as $\gamma_1 < \gamma_2$ and regardless of the intrinsic delay of the buffers. Therefore, there is no optimum delay for any positive values of m_1 and m_2, unless one of them is set to zero, which makes the buffer tree use only one buffer type. The same reasoning is applicable for the case $\gamma_1 > \gamma_2$ and for more buffer types as well.

■

APPENDIX C

Finding the best buffer

Theorem. The Best Buffer is the one with the smallest μ in the buffer library.

Proof. The necessary and sufficient condition to have the total delay of the buffer tree at its minimum is to have $\left(\beta_0 \gamma_T \left(\beta_b \gamma_b \right)^k \right)^{\frac{1}{k+1}}$ at its minimum in the original delay equation:

$$D = k\alpha_b + (k+1)\left(\beta_0 \gamma_T \left(\beta_b \gamma_b \right)^k \right)^{\frac{1}{k+1}}$$

This is true regardless of the sum of intrinsic delays $k\alpha_b$ [1] .Since μ is defined as:

$$\mu = \left(\beta_0 \gamma_T \left(\beta_b \gamma_b \right)^k \right)^{\frac{1}{k+1}}$$

It is clear that the delay equation is minimum for any buffer that yields the minimum μ. This buffer is defined as the *best buffer*.

[1] See appendix B: Proof of the Best Buffer Existence.

REFERENCES

[AHO et al, 1983] AHO, A. H., HOPCROFT, J.E., ULLMAN, J.D.,(1983), *Data Structures and Algorithms*, Chapter 10, Addison-Wesley.

[ALPERT et al, 2000] ALPERT, C. J., GANDHAM, R. G., NEVES, J. L., QUAY, S. T. (2000), *Buffer library selection*, IEEE International Conference on Computer Design, p. 221-226.

[ALPERT et al, 2001] ALPERT, C. J., KAHNG, A. B., LIU, B., MANDOIU, I., ZELIKOVSKY, A.,(2001), *Minimum-buffered routing of non-critical nets for slew rate and reliability control*, Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, p. 408-415.

[ALPERT and DEVGAN, 1997] ALPERT, C. J., DEVGAN, A., (1997), *Wire segmenting for improved buffer insertion*, IBM Austin Research Laboratory, Austin, TX 78758, Design Automation Conference (DAC), p. 588-593.

[AMOURA and MAILHOT, w. d.] AMOURA, A., MAILHOT, F., (w. d.) *Private communication*.

[BELLMAN, 1957] BELLMAN, R. E., (1957), Dynamic Programming, Princeton University Press.

[BERMAN et al, 1989] BERMAN, C.L., CARTER, J.L., DAY, K.F., (1989), *The fanout problem: from theory to practice*, SEITZ, C. L., editor, Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference, MIT Press, p. 69-89.

[BOHR, 1995] BOHR, M. T., (1995), *Interconnect Scaling - The real limiter to high performance ULSI*, Technical Digest, IEEE International Electronic Devices Meeting, p. 241-244.

[CARRAGHER and CHENG, 1995] CARRAGHER, R. J., CHENG, C. -K., (1995), *Simple tree-construction heuristics for the fanout problem*, Proceeding of International Conference on Computer-Aided Design, p. 671-679.

[CHEN et al, 2002] CHEN, W., PEDRAM, M., BUCH, P., (2002), *Buffered routing tree construction under buffer placement blockage*, Proceeding of the 2002 Conference on Asia South Pacific Design, p.381.

[CHU and WONG, 1997] CHU, C.C.N., WONG, D. E., (1997), *Closed form solution to simultaneous buffer insertion/sizing and wire sizing*, International Symposium on Physical Design, p. 192-197.

[CHU and WONG, 1999] CHU, C.C.N., WONG, D. E., (1999), *A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing*, IEEE Transaction on Computer Aided Design of Integrated Circuit System, vol. 18, no. 6, p. 787-798.

[CONG et al, 1993] CONG, J., LEUNG, K.S., ZHOU, D. (1993), *Performance-driven interconnect design based on distributed RC delay model*, Proceeding of ACM/IEEE Design Automation Conference, p. 606-611.

[CONG and YUAN, 2000] CONG, J., YUAN, X., (2000), *Routing tree construction under fixed buffer locations*, Proceeding of Design Automation Conference, p. 379-384.

[CORMEN et al, 2001] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., (2001), *Introduction to algorithms*, second edition, p. 323-324.

[DHAR and FRANKLIN, 1991] DHAR, S., FRANKLIN, M. A., (1991), *Optimum buffer circuits for driving long uniform lines*, IEEE J. Solid-State Circuits, vol. 26, no. 1, p. 32-40.

[ELMORE, 1948] ELMORE, W. C., (1948), *The transient response of damped linear networks with particular regard to wideband amplifiers*, Journal of Applied Physics, vol. 19, p. 55-63.

[ERDOS, 1946] ERDOS, P., (1946), *On sets of distances of n points*, American Mathematical Monthly no.53, p. 248-250.

[GAUSS, 1809] GAUSS, C.F., (1809), *Theoria motus corporum coelestium in sectionibus conicis solem ambientum (theory of motion of the celestial bodies moving in conic sections around the sun)*.

[HRKIC and LILLIS. 2002] HRKIC, M., LILLIS, J., (2002), *S-tree: A technique for buffered routing tree synthesis*, New Orleans, LA, Proceeding of ACM/IEEE Design Automation Conference, p. 578–583.

[HUANG et al. 2003] HUANG, H., -D., LAI, M., WONG, D. F., GAO, Y., (2003), *Maze routing with buffer insertion under transition time constraints*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 1. p. 91-96.

[ISMAIL et al, 1999] ISMAIL, Y. I. , FRIEDMAN, E. G., NEVES, J. L. (1999), *Equivalent Elmore Delay for RLC Trees*, Proceedings of the 36th ACM/IEEE conference on Design automation, p. 715-720.

[JIANG and CHANG, 2004] JIANG, I. H., CHANG, Y. –W., (2004), *Simultaneous floorplan and buffer-block optimization,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 5, p. 694.

[KANG et al, 1997a] KANG, M., DAI, W. W.-M., DILLINGER, T., LAPOTIN, D., (1997), *Delay bounded buffered tree construction for timing driven floorplanning,* Proceeding IEEE/ACM Internation Conference of Computer-Aided Design, p. 707-712.

[KANG et al, 1997b] KANG, M., DAI, W. W.-M., DILLINGER, T., LAPOTIN, D., (1997), *Timing-driven floorplanning with intermediate buffer insertion,* Technical Report: UCSC-CRL-97-03, University of California at Santa Cruz, California, USA.

[KNUTH , 1998] KNUTH, D. E., (1998), *The Art of Computer Programming, Volume 3 - Sorting and Searching*, Addison-Wesley, 2nd edition.

[LAI and WONG, 2000] LAI, M. WONG, D. F., (2000), *Maze routing with buffer insertion and wire sizing,* ACM/IEEE Design Automation Conference, p. 374-378.

[LAND and DOIG, 1960] LAND, A. H., DOIG, A.,(1960), *An automatic method for solving discrete programming problems,* Econometrica, 28, p. 497-520.

[LI, 1992] LI, W., (1992), *Random texts exhibit Zipf's-law-like word frequency distribution,* IEEE Transaction on Information Theory, p.1842-1845.

[LILLIS et al.1995] LILLIS, J., CHENG, C.-K., LIN, T.-T. Y., (1995), *Optimal wire sizing and buffer insertion for low power and a generalized delay model*, IEEE Journal of Solid-State Circuits, p. 437-447.

[LILLIS et al, 1996a] LILLIS, J., CHENG, C.-K, Y LIN, T.-T., (1996), *New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing,* Proceeding of 33rd ACM/IEEE Design Automation Conference, p. 395-400.

[LILLIS et al, 1996b] LILLIS, J., CHENG, C.-K, Y LIN, T.-T., (1996), *Simultaneous routing and buffer insertion for high performance interconnect*, Proceeding of the Sixth Great Lakes Symposium on VLSI, p. 148-153.

[MAILHOT, 2005] MAILHOT, F., (2005), *Computers architecture II– GEI 431,* course material offered at the Department of Electrical and Computer Engineering, University of Sherbrooke, Quebec, Canada.

[MICHALEWICZ and FOGEL , 2000] MICHALEWICZ, Z., FOGEL, D. B.,(2000), *How to Solve It: Modern Heuristics*, Springer-Verlag, Chapter 3.

[MITCHELL, 1997] MITCHELL, T. M.,(1997), *Machine Learning*, WCB McGraw-Hill.

[OKAMOTO and CONG , 1996a] OKAMOTO, T., CONG, J., (1996), *Buffered Steiner tree construction with wire sizing for interconnect layout optimization*, Proceeding of IEEE/ACM International Conference on Computer-Aided Design, p. 44–49.

[OKAMOTO and CONG, 1996b] OKAMOTO, T., CONG, J., (1996), *Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion*, Fifth ACM/SIGDA Physical Design Workshop, p.1-6.

[OZMAY, 2006] OZMAY, E., (2006), *Plasmonics: merging photonics and electronics at nano-scale dimensions,* In Science, vol. 311. no. 5758, p. 189 – 193.

[RABBANI and MAILHOT, 2007] RABBANI, A. H., MAILHOT, F., (2007), *Efficient buffer tree construction using mixed branch-and-bound and dynamic programming techniques*, Montreal, International Symposium on Signals, Systems and Electronics, p. 395 – 398.

[SALEK et al, 1999] SALEK, A. H., LOU, J., PEDRAM, M., (1999), *MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search*, Proceeding of the 36th ACM/IEEE Conference on Design Automation, p. 472 – 478.

[SHI and ZI, 2005] SHI, W., LI, Z., (2005), *A fast algorithm for optimal buffer insertion*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 6. p. 879-891.

[SHI et al, 2004] SHI, W., LI. Z., ALPERT. C. J., (2004), *Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost*, Yokohoma, Japan, Proceeding of the 2004 Conference on Asia South Pacific Design Automation, p. 609-614.

[SMITH, 1998] SMITH, M. J. S., (1998), *Application-specific integrated circuits*, Addison Wesley VLSI System Series, p. 1026.

[TANG et al, 2001] TANG, X., TIAN, R., XIANG, H., WONG, D.F., (2001), *A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints,* Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design, p. 49-56.

[TANG and WONG, 2004] TANG, X., WONG, M., (2004), *Tradeoff routing resource, runtime and quality in buffered routing*, Proceeding of the 2004 Conference on Asia South Pacific Design Automation, p. 430-433.

[TELLEZ and SARRAFZADEH 1997] TELLEZ, G. E., SARRAFZADEH, M., (1997), *Minimal buffer insertion in clock trees with skew and slew rate constraints*, IEEE Transactions on Computer-Aided Design, p.333–342.

[TOUATI, 1990] TOUATI, H., (1990), *Performance-oriented technology mapping*, University of California, Berkeley, Technical Report Memorandum UCB/ERL M90/109.

[VAN GINNEKEN, 1990] VAN GINNEKEN, L.P.P.P., (1990), *Buffer placement in distributed RC-tree networks for minimal Elmore delay*, Proceeding of IEEE International Symposium on Circuits and Systems, p. 865–868.

[VOGEL and WONG, 2006] VOGEL, S. WONG, D.F.,(2006), *Closed form solution for optimal buffer sizing using the Werierstrass elliptic function*, Proceeding of the 2006 Conference on Asia South Pacific Design Automation, p. 315-319.

[WANG et al, 2005] WANG, K., RAN, Y., JIANG, H., MAREK-SADOWSKA, M.,(2005) *General skew constrained clock network sizing based on sequential linear programming*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 5. p. 773-782

[ZHOU et al, 2000] ZHOU, H., WONG, D. F., LIU, I. M., AZIZ, A., (2000), *Simultaneous routing and buffer insertion with restrictions on buffer locations,* IEEE Transaction Computer Aided Design Integrated Circuits, vol. 19, no. 7, p. 819–824.

[ZHU, 1995] ZHU, Q. (1995), *Chip and Package Co-Synthesis of Clock Networks*, PhD thesis, University of California, Santa Cruz, California.