Efficient Buffer Tree Creation Using Mixed Branch and Bound and Dynamic Programming Techniques

Amir Hossein Rabbani,
*Frederic Mailhot
Département de Génie Électrique et Génie Informatique
Université de Sherbrooke
Sherbrooke, J1K 2R1, Canada

Abstract- In order to reduce circuit delay and help driving large fan-out, buffer insertion needs to be performed during logic and physical synthesis. This optimization activity is often done based on dynamic programming. We introduce a new and effective method in which we combine Dynamic Programming and Branch and Bound techniques in order to improve the run time and quality of the buffer tree as compared to common methods. We solve the problem for the specific case of buffering balanced trees, where all loads have identical required time and input load capacitance. We provide the underlying concepts for a generalized version of our algorithm, where a set of different load capacitances and required times is given.

Keywords: Buffer Circuits, Circuit Synthesis, Dynamic Programming,

I. INTRODUCTION

Buffering is a technique used during logic and physical synthesis. It is applied to reduce circuit delay and allow logic gates to drive large fan-outs. Since the significant work of Van Ginneken [1] which proposed a dynamic programming algorithm for inserting buffers and that of Touati [6] , 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 a NP-complete problem [2,3], under certain circumstances exhaustive search techniques can be more effective than dynamic programming, in order to cover the space of possible solutions and result in better buffer trees.

Previously, a Branch and Bound algorithm was proposed to solve the specific case of buffering where all loads have identical required times and input capacitances [4]. The method is called Balanced Buffering and yields the fastest buffered tree together with the best topology. Given a source output resistance and a set of identical sinks, it 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. However, the technique can require important runtimes, particularly in the presence of large fan-outs and large buffer libraries [5].

We have identified an efficient structure for solving balanced buffering problems which allows faster search time with a reasonable memory footprint. To that effect, we combine the concepts of Dynamic Programming with those of Branch and Bound algorithms. Compared to a simple Branch and Bound method where no sub-problem solution is kept, our method saves the sub-problem results and reuses these results for quickly solving similar sub-problems. While storing all sub-solutions would require much more memory than a simple Branch and Bound technique, our method has shown only a small increase in memory usage during our tests. This is due to our use of Dynamic Programming which regroups similar sub-solutions. Our hybrid method has shown runtimes up to 6000 times faster than the original algorithm.

The remainder of the paper is organized as follows: Section 2 defines the Balanced Buffering problem and discusses how one can improve the regular Branch and Bound buffering algorithm [5]. Section 3 covers the theoretical and technical elements of the method and section 4 presents the experimental results.

II. PROBLEM ANALYSIS

Branch and Bound versus Dynamic Programming

The Branch and Bound algorithm enumerates every possible solution of a balanced buffering problem, calculating only those solutions allowed by the bound. Thus, one expects to achieve a high quality solution, potentially at the cost of requiring large runtimes. A Branch and Bound method can be very effective whenever we encounter independent sub-problems, where solving one does not affect the results of solving the others. It also needs a tight bound in order to effectively filter the potentially enormous space of possible solutions. However, if the complete problem encompasses a large set of identical sub-problems, the Branch and Bound method can waste a significant amount of runtime, repeatedly recalculating the same sub-problems. In contrast, a dynamic programming technique can be applied whenever common sub-problems form the basic structure of a problem. Using a structure tailored to dynamic programming, one can solve the problem efficiently in terms of both memory usage and runtime.

1-4244-1449-0/07/$25.00 © 2007 IEEE

In solving the balanced buffering problem, where there exist many similar sub-solutions and also many redundant solutions, we can profit from the advantages of both Branch and Bound and dynamic programming methods. We apply the idea of solution-reuse from dynamic programming to avoid unnecessary recalculations and when appropriate, utilize a bound to prune away non-promising sub-problems. The necessary bound has been introduced elsewhere [4,5] and will be briefly discussed in section 3. In order to have an effective combination of Branch and Bound and dynamic programming techniques, we define an efficient structure for handling balanced buffering problems and sub-problems.

Efficient Structure

Our balanced buffering method has some similarity with that of Van Ginneken's technique [1]. The main difference is in handling the issue of topology. Van Ginneken's approach solves the problem for a given topology, whereas in our balanced buffering system the best topology is being searched as well as the best buffer arrangements. Thus, we modify the buffering problem definition and use the description:

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 predefined buffer library.

It is clear that there always exists at least one best buffer tree (as defined above) for a given source and 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 we explore all branching factors. For every sub-problem, the best branching factor is achieved by exploring every possible set of sub-trees, using their solutions to evaluate the total delay of the sub-problem. That is where we need Branch and Bound to explore only promising sub-trees.

The 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. We resolve this issue 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, we preserve the optimality of the Branch and Bound method while increasing search-speed and by dynamic programming.

Mixed Method Efficiency in Solving Balanced Buffering

Due to the symmetrical structure of sub-trees in a balanced buffering problem, we encounter many similar sub-problems. Therefore, we expect our mixed method to reuse many common solutions while searching for the best balanced buffer tree. As a result, sub-problems reuse will lead to better runtime, while keeping memory usage low, as shown in experimental results appearing in section IV. However, it is important to note that the efficiency of mixing the two techniques (Balanced Buffering 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, we currently traverse the sub-problems based on their ideal buffer tree delay. Therefore, for every sub-problem, we compute the delay of the ideal buffer tree, and then traverse them in the order of smallest ideal delay. This technique is generally satisfactory, as shown by the experimental results.

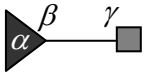
The second factor to consider is the way sub-problems are distributed in the search space. The number of sub-problems and the way their dependencies are established are directly influenced by the timing properties of buffers. In [5] we show that the order of buffer arrangements in the best buffer tree obeys certain physical constraints. Due to additional memory allocation for saving solutions, the gain of using a mixed method must be carefully studied to guarantee a good payback.

III. Theoretical and Technical Elements

The Bound

In order to implement the Branch and Bound algorithm, an effective bound is necessary. In [4,5], the underlying delay model used for the bound calculation is the simplistic Elmore delay formula, where no wire delay is taken into account. Although modern buffer characterization is based on much more complex timing models, the simple Elmore delay can always be extracted and will produce an optimistic bound (i.e. smaller ideal delay), the impact of which will be on runtime but not on quality. For a buffer being directly connected to a load, the Elmore delay is:

$$D = \beta\gamma + \alpha \qquad (1)$$



where β is the buffer output resistance, γ is the load input capacitance and α is the intrinsic delay of the buffer.

As mentioned before, the bound can be defined as the minimum delay of an ideal balanced buffer tree for a given source and a set of sinks. It is calculated as follows:

$$Bound = D_{current} + \alpha_b + D_{\min} \quad (2)$$

where $D_{current}$ is the actual buffer tree delay up to the root of the sub-tree for which we are going to do balanced buffering, α_b is then the intrinsic delay of this buffer and Dmin is the minimum ideal delay achievable for the sub-tree.

The minimum ideal delay is defined as follows [4,5]:

$$D_{\min} = \mu\left(1 + Ln\left(\frac{\beta_{source}\gamma_T}{\mu}\right)\right) \quad (3)$$

where β_{source} is the output resistance of the source gate, γ_T is the total input capacitance of the loads and μ is:

$$\mu = \beta_b\gamma_b e^{\left(\frac{\alpha_b}{\mu}+1\right)} \quad (4)$$

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. The best buffer is the one with the smallest μ in a given buffer library. One can systematically find this best buffer before starting to solve the problem and use its values in calculating the bound. The ideal delay is then safe to use as a bound as equation (3) assumes ideal conditions that cannot occur in real problems. In particular, it assumes continuous rather than discrete values for the number of levels and branches in the ideal buffer tree. The bound is therefore used to eliminate non-promising sub-problems whenever it shows an ideal delay larger than the best real delay achieved so far.

Search Space

In our mixed method, a typical search space consists of 3 groups of solutions:

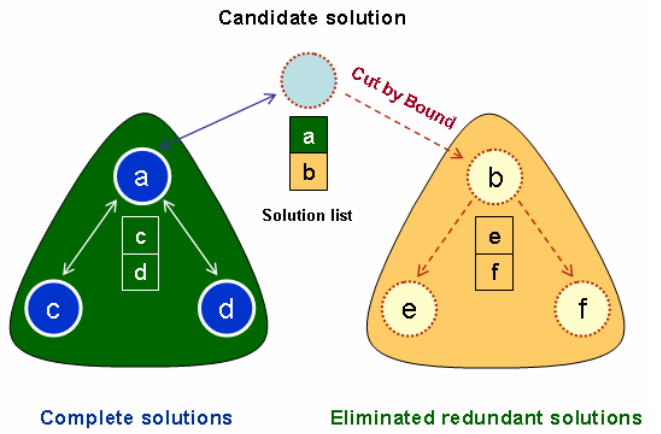
a) Eliminated redundant solutions: those which are cut by the bound. Their solution lists are not built.

b) Complete solutions: those whose results are already available and are built of previously calculated sub-solutions. We do not recalculate their sub-solutions; instead, we use the best solution in their solution lists to avoid redundant calculations.

c) Candidate solutions: those which have both complete and redundant solutions in their solution lists. Although we cannot use them to obtain the best solution, we save their solution lists. This helps them being solved faster in further references.

Fig.[1] illustrates the correlation between these 3 solution types.

Fig. 1. 3 solution types. Only the solution lists of candidate and complete solutions are saved.



Pruning Function

There is often a trade-off between memory usage and runtime. Whereas we improve the runtime of balanced buffering by saving solution lists, we demand a remarkably higher amount of memory in comparison with a simple Branch and Bound. To resolve this drawback and help keeping the search space small, we prune a solution list by deleting those solutions which are dominated by a complete solution. In other words, if the ideal delay of an unsolved solution is already worse than the real delay of a complete solution, it can never be better than that complete solution, and hence becomes redundant. Our studies show that pruning a solution list has the major role in reducing memory usage of our algorithm.

Efficient Speed-Up Technique

Candidate solutions are frequently recalculated during search space traversal. In order to have a faster runtime for this group of solutions, we can insert additional data in their solution lists. This data is the "current delay" used in calculating the bound for a sub-problem (equation 2). At the time of next reference to an already explored candidate solution, we compare the last "current delay" and the new one. If the recent "current delay" is greater than or equal to the old "current delay", solving the sub-problem does not lead to any better solution. Otherwise, we update the "current delay" and recalculate the sub-problem. We call this technique "The Enhanced Mixed Method" and we will show its considerable effect on runtime improvement in section 4.

IV. Experimental Results

Our algorithm is implemented in C++ and can run with and without the dynamic programming enhancement. We run the program on a Pentium IV computer (CPU 3.4 GHz) and for different fan-outs, ranging from 100 to 1000. We apply the same buffer library for all problems and it consists of 6 non-inverting buffers. The intrinsic delay, output resistance and

input capacitance for the buffers range from 3-40 ps, 0.05-0.5 Ω and 70-300 fF. The input capacitance of the loads is set to 500 fF and the output resistance of the driving source is set to 0.5 Ω. The results of solving such problems are shown in Table.1. Using the mixed method, we observe 2 to 3 times faster runtime than a simple Branch and Bound method. Utilizing the enhanced mixed method for candidate solutions, we achieve a runtime up to 6000 times faster than a simple Branch and Bound, which is a significant contribution to the search speed of our mixed method. As shown in Table.1, the mixed method operates with a reasonably low memory usage, while by pruning the solution lists its memory consumption becomes even more efficient.

V. Conclusion

While buffering in general is a NP-complete problem, we have shown that there exist efficient ways to find the optimum delay solutions to balanced buffering problems. With a combination of effectively exhaustive yet efficient search techniques, and memory-reuse concepts of Dynamic Programming, we can find optimum solutions to that class of buffer tree problems. The idea of having such a mixed method can be applied to solve any similar hybrid problem demonstrating both Dynamic Programming optimal structure and the possibility of having a mathematically computable bound to implement the Branch and Bound algorithm. By this, one can profit from both the solution quality of the Branch and Bound method, and fast yet reasonably low in memory consumption techniques of Dynamic Programming. On the other hand, having a reliable bound, we can also improve a mere Dynamic Programming algorithm by analytically selecting the promising sub-problems to be solved, and thus save time and memory.

Studies are in progress to apply our hybrid algorithm to generalized buffer tree problems, where the loads are not identical in terms of required time and input capacitance.

References

[1] L.P.P.P. Van Ginneken.(1990) "Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay". In Proc. IEEE Intl. Symp. Circuits and Systems, pages 865–868.

[2] C.L. Berman, J.L. Carter and K.F. Day (1989) "The Fanout Problem: From Theory to Practice". In C. L. Seitz, editor, Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference, pp. 69-89. MIT Press

[3] W. Shi, Z. Li, and C. J. Alpert, (2004) "Complexity Analysis and Speedup Techniques for Optimal Buffer Insertion with Minimum Cost", ASPDAC, pp. 609–614.

[4] F. Mailhot and A. Amoura, "Private Communication"

[5] A. Rabbani, A. Amoura and F. Mailhot, "A Branch and Bound Approach to the Balanced Buffering Problem", to be published.

[6] H. Touati. "Performance-Oriented Technology Mapping." PhD thesis, U. C. Berkeley, November 1990. Memorandum UCB/ERL M90/109

TABLE I
RUNTIME AND MEMORY USAGE FOR DIFFERENT METHODS

Fanout	Runtime (seconds)			Memory Usage (bytes)			
	Branch and Bound	Mixed Method	Enhanced Mixed Method	Branch and Bound	Mixed Method		Enhanced Mixed Method (with pruning)
					Without Pruning	With Pruning	
100	488	203.266	0.937	Negligible	407,640	246,780	246,780
200	1936.938	861.703	1.719		644,136	404,392	404,392
300	7665.750	3024.219	3.343		1,085,012	702,100	702,100
500	14466.094	6483.219	4.312		1,428,480	932,096	932,096
1000	59077.875	26324.812	9.406		2,336,120	1,586,452	1,586,452