

Making Fast Buffer Insertion Even Faster Via Approximation Techniques *

Zhuo Li¹, C. N. Sze¹, Charles J. Alpert^{*}, Jiang Hu¹, and Weiping Shi¹

¹Dept. of Electrical Engineering, Texas A&M University, College Station, TX 77843

^{*} IBM Austin Research Lab, 11501 Burnet Road, Austin, TX, 77858

Abstract— As technology scales to 0.13 micron and below, designs are requiring buffers to be inserted on interconnects of even moderate length for both critical paths and fixing electrical violations. Consequently, buffer insertion is needed on tens of thousands of nets during physical synthesis optimization. Even the fast implementation of van Ginneken’s algorithm requires several hours to perform this task. This work seeks to speed up the van Ginneken style algorithms by an order of magnitude while achieving similar results. To this end, we present three approximation techniques in order to speed up the algorithm: (1) aggressive pre-buffer slack pruning, (2) squeeze pruning, and (3) library lookup. Experimental results from industrial designs show that using these techniques together yields solutions in 9 to 25 times faster than van Ginneken style algorithms, while only sacrificing less than 3% delay penalty.

I. INTRODUCTION

Owing to the tremendous drop in VLSI feature size, a huge number of buffers are needed for achieving timing objectives and fixing electrical violations for interconnects. It is stated in [1] that the number of buffers will rise dramatically, for example, reaching about 15% of the total cell count for intrablock communications for 65nm technology. The number becomes 35% and 70% for the 45nm and 32nm technology nodes respectively. Although we are not sure whether the number of 70% will finally be reached, hundreds of thousands of buffers can be found in today’s ASICs. For example, Osler [2] presents an existing chip with 426 thousand buffers which occupy 15% of the available area. Therefore, both the complexity and importance of buffer insertion is increasing in an even faster pace.

The increasing number of buffers cause various design problems such as congestion and space management. Despite those design problems, it is also a challenge to insert them efficiently and automatically. Even a buffer insertion tool that can process five nets a second requires around 70 hours to process one-hundred thousand nets. An order of magnitude speedup in buffer insertion technologies could enable this task to be accomplished in less than an hour while it also enables more design iterations and faster timing closure.

In 1990, van Ginneken [3] proposed a classical buffer insertion algorithm. The algorithm has time complexity $O(n^2)$,

where n is the number of candidate buffer locations. Recently, Shi and Li [4] improved the time complexity to $O(n \log n)$ for 2-pin nets and $O(n \log^2 n)$ for multi-pin nets. Several works have built upon van Ginneken’s algorithm to include wire sizing [6], higher-order delay models [7, 8], simultaneous tree construction [9, 10, 11, 12], and noise constraints [13].

However, van Ginneken’s algorithm does not control buffering resources while it only focuses on obtaining the optimal slack. In practice, inserting 30 buffers to fix a slew constraint or to meet a delay target when 3 buffers may suffice is not acceptable as it will accelerate the buffer explosion crisis. Also, people frequently want to find the cheapest solution that meets the timing target, not necessarily the optimal solution in terms of maximum slack. Lillis *et al.* [6] presented an implementation to control resource utilization. Towards this direction, we use a simple cost function to find the solution with the minimum number of buffers that meets the delay target and fixes the slew constraint. Also, instead of a single buffer type, we consider a discrete set of non-inverting and inverting buffers of various power levels, which is also mentioned in [6].

With the buffer library and cost consideration, van Ginneken’s algorithm becomes practical yet more runtime intensive. The work in [14] has proved that minimizing buffering resource in buffer insertion is NP-complete. In this work, we propose three speedup techniques to the mentioned variation of generalized buffer insertion algorithm.

1. **Aggressive Pre-buffer Slack Pruning (APSP)**. When considering candidate buffer solutions at a node, it can be concluded that there must exist a minimum resistance driving this node, which leads to additional delay. This delay can be factored in during pruning to prune more aggressively than van Ginneken’s algorithm allows. One can even increase this resistance value to prune more aggressively but potentially sacrifice optimality.
2. **Squeeze Pruning (SqP)**. When examining three sorted candidates, one may be able to guess that the middle one will soon become dominated by the one just below or just above it in the candidate solution list. By determining this before the candidate actually becomes dominated, squeeze pruning allows it to be removed earlier.
3. **Library Lookup (LL)**. During van Ginneken style buffer insertion, one considers inserting each buffer from a library for a given candidate. Instead, Library Lookup considers just the inverting and non-inverting buffer from the

*Research of Z. Li and W. Shi was supported in part by NSF grants CCR-0098329, CCR-0113668, EIA-0223785 and ATP grant 512-0266-2001. Research of C. N. Sze and J. Hu was supported in part by SRC under contract 2003-TJ-1124.

library that yields the best delay. These are determined from a pre-computed lookup table.

Different from the work of [5, 14], which emphasizes more on large and huge nets, our techniques are more effective on small and medium nets which are the majority in most chip designs. Our experiments on thousands of nets from industry designs show that when we combine these three techniques, we are able to gain a factor of $9\times$ to $25\times$ speedup over traditional buffer insertion algorithm while the slack only degrades by 2-3% on average. These techniques can be easily integrated with the current buffer insertion engine which considers slew, noise and capacitance constraints. Consequently, we believe these techniques are essential to embed in a physical synthesis buffer insertion system.

II. PRELIMINARIES

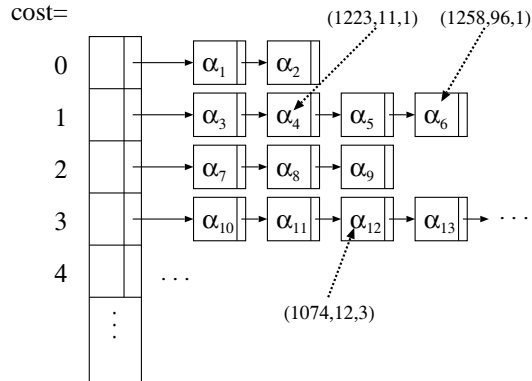
Given a Steiner tree with a set of buffer locations (namely the internal nodes), the buffer insertion problem is to decide whether a buffer is inserted at each internal node such that the required arrival time (RAT) at the source is maximized. In the classical buffer insertion algorithm [3, 6], candidate solutions are generated and propagated from the sinks toward the source. Each candidate solution α is associated with an internal node in the tree and is characterized by a 3-tuple (q, c, w) . The value q represents the required arrival time; c is the downstream load capacitance; and w is the cost summation for the buffer insertion decision.

Initially, a single candidate (q, c, w) is assigned for each sink where q is the sink RAT, c is the load capacitance and $w = 0$. When the candidate solutions are propagated from a node to its parent, all three terms are updated accordingly. At an internal node, a new candidate is generated by inserting a buffer. At each Steiner node, two sets of solutions from the children are merged. Finally at the source, the solutions with max q are selected.

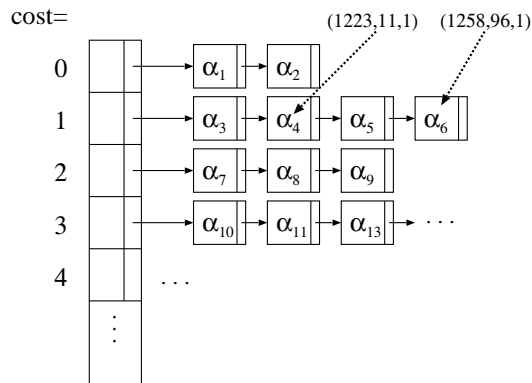
The candidate solutions at each node are organized as an array of linked lists as shown in Fig. 1. The solutions in each list of the array have the same buffer cost value $w = 0, 1, 2, \dots$. The polarity is handled by maintaining two arrays of candidate solutions. In buffer insertion algorithm, a solution can be pruned only if it is redundant, i.e., there exists another solution that is better in slack, capacitance and buffer cost. More specifically, for two candidate solutions $\alpha_1 = (q_1, c_1, w_1)$ and $\alpha_2 = (q_2, c_2, w_2)$, α_2 dominates α_1 if $q_2 \geq q_1$, $c_2 \leq c_1$ and $w_2 \leq w_1$. In such case, we say α_1 is redundant and has to be pruned. For example, in Fig. 1(a), assume $\alpha_4 = (1223ps, 11fF, 1)$ and $\alpha_{12} = (1074ps, 12fF, 3)$, α_{12} is dominated by α_4 and is then pruned. The data structure after pruning is shown in 1(b). After pruning, every list with the same cost is a sorted in terms of q and c .

A buffer library is a set of buffers and inverters, while each of them is associated with its driving resistance, input capacitance, intrinsic delay, and buffer cost. During optimization, we wish to control the total buffer resources so that the design is not over-buffered for marginal timing improvement. While total buffer area can be used, to the first order, the num-

ber of buffers provides a reasonably good approximation for the buffer resource utilization. Indeed, we use the number of buffers since it allows a much more efficient baseline van Ginneken implementation. Note that, our techniques presented in this paper can be applied on any buffer resources model, such as total buffer area or power.



(a) The basic data structure storing candidate solutions



(b) After pruning

Fig. 1. Examples of data structure and pruning

At the end of the algorithm, a set of solutions with different cost-RAT tradeoff is obtained. Each solution gives the maximum RAT achieved under the corresponding cost bound. Practically, we choose neither the solution with maximum RAT at source nor the one with minimum total buffer cost. Usually, we would like to pick one solution in the middle such that the solution with one more buffer brings marginal timing gain. In our implementation, we use the following scheme namely the “10ps rule”. For the final solutions sorted by the source’s RAT value, we start from the solution with maximum RAT and compare it with the second solution (usually it has one buffer less). If the difference in RAT is more than 10ps, we pick the first solution. Otherwise, we drop it (since with less than 10ps timing improvement, it does not worth an extra buffer) and continue

to compare the second and the third solution. Of course, instead of $10ps$, any time threshold can be used when applying to different nets.

III. AGGRESSIVE PRE-BUFFER SLACK PRUNING (APSP)

In most of previous works [3, 6], a candidate solution is pruned out if there is another solution with less capacitance, larger slack and less buffer cost. Such pruning is based on the information at the node being processed. However, the solutions at this node is always propagated toward the source and we can prune out more potentially inferior solutions by anticipating the upstream information. More specifically, the solutions at the node is always driven by an upstream resistance (such as buffer or driver resistance) of at least R_{min} . The pruning based on anticipated upstream resistance is called the pre-buffer slack pruning.

Pre-buffer Slack Pruning (PSP): For two non-redundant solutions (q_1, c_1, w) and (q_2, c_2, w) , where $q_1 < q_2$ and $c_1 < c_2$, if $(q_2 - q_1)/(c_2 - c_1) \geq R_{min}$, then (q_2, c_2, w) is pruned.

For example, in Fig. 1(b), we assume $R_{min} = 1k\Omega$. For two non-dominating candidates $\alpha_4 = (1223ps, 11fF, 1)$ and $\alpha_6 = (1258ps, 96fF, 1)$, since $1258 - 1000 \times 0.096 = 1162 < 1223 - 1000 \times 0.011 = 1212$, α_6 is pruned since we certainly predict α_6 will be dominated by α_4 when the bottom-up process continues. In other words, the extra $85fF$ of capacitance will add at least $85ps$ of upstream delay eventually. Thus, the $25ps$ slack advantage of α_4 is overshadowed by its weakness of larger capacitance.

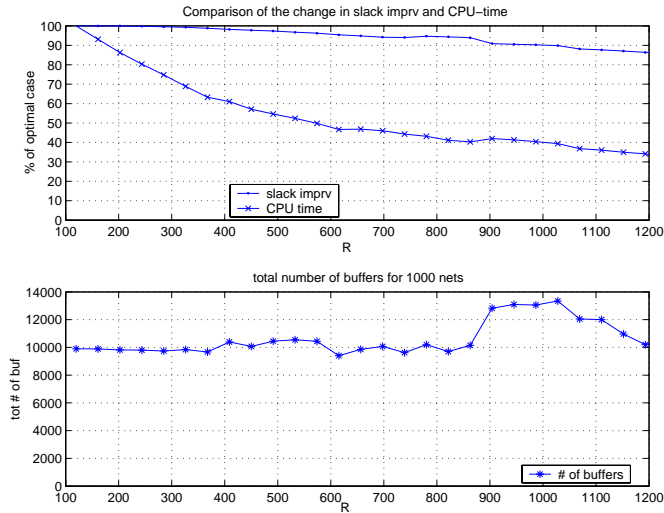


Fig. 2. The speed-up and solution sacrifice of APSP in 1000 nets

The pre-buffer slack pruning technique was first proposed in [5] and it has been shown that the technique is very effective and produces optimal results. The detail proof and discussion can be found in [5, 14].

An interesting observation is that if we use a resistance R which is larger than R_{min} value, obviously more solutions are pruned and the algorithm becomes faster. However, it will sacrifice the solution quality. This technique is referred to as the

aggressive pre-buffer slack pruning - APSP since it does more aggressive pruning than the technique in [5]. In order to investigate the relationship of algorithm speed-up and solution sacrifice for APSP, we have performed a set of experiments on 1000 industrial nets (with a buffer library consisting of 24 buffers). The comparison is shown in Fig. 2. The figure shows the degradation in slack and the decrease in CPU time as a function of resistance. When $R = 120\Omega$, this is the minimum resistance value which still yields the optimal solution. As R increases, the CPU time drops much more sharply than the slack. For example, one can get a 50% speedup for less than 5% slack degradation when $R = 600\Omega$. Also, note from the bottom chart that the number of buffers stays fairly stable until R gets quite large. The promising experiment results show that by using APSP, a tiny sacrifice in solution quality can bring a huge speed-up in the van Ginneken's algorithm.

IV. SQUEEZE PRUNING (SQP)

The basic data structure of van Ginneken style algorithms is a sorted list of non-dominated candidate solutions. Both the pruning in van Ginneken style algorithm and the pre-buffer slack pruning are performed by comparing two neighboring candidate solutions at a time. However, more potentially inferior solutions can be pruned out by comparing three neighboring candidate solutions simultaneously. For three solutions in the sorted list, the middle one may be pruned according to the squeeze pruning defined as follows.

Squeeze Pruning: For every three candidate solutions (q_1, c_1, w) , (q_2, c_2, w) , (q_3, c_3, w) , where $q_1 < q_2 < q_3$ and $c_1 < c_2 < c_3$, if $(q_2 - q_1)/(c_2 - c_1) < (q_3 - q_2)/(c_3 - c_2)$, then (q_2, c_2, w) is pruned.

For a two-pin net, consider the case that the algorithm proceeds to a buffer location and there are three sorted candidate solutions with the same cost that correspond to the first three candidate solutions in in Fig. 3(a). According to the rationale in pre-buffer slack pruning, the q - c slope for two neighboring candidate solutions tells the potential that the candidate solution with smaller c can prune out the other one. A small slope implies a high potential. For example, (q_1, c_1, w) has a high potential to prune out (q_2, c_2, w) if $(q_2 - q_1)/(c_2 - c_1)$ is small. If the slope value between the first and the second candidate solutions is smaller than the slope value between the second and the third candidate solutions, then the middle candidate solution is always dominated by either the first candidate solution or the third candidate solution. Squeeze pruning keeps optimality for a two-pin net. After squeeze pruning, the solution curve in (q, c) plane is concave as shown in Fig. 3 (b).

For a multi-sink net with Steiner nodes, squeeze pruning can not keep optimality since each candidate solution may merge with different candidate solutions from the other branch and the middle candidate solution in Fig. 3 (a) may offer smaller capacitance to other candidate solutions in the other branch. Squeeze pruning may prune out a post-merging candidate solution that is originally with less total capacitance. Therefore, the final solution may be sub-optimal. However, squeeze pruning only causes little degradation on the solution quality since

it is performed for each set of solutions with the same cost and the capacitance under-estimation effect is alleviated.

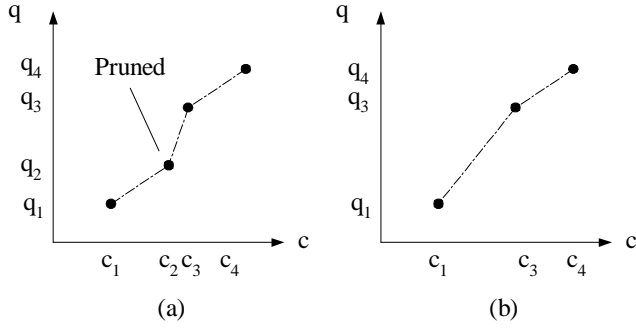


Fig. 3. Squeeze Pruning Example. (a) The solution curve in (q, c) plane before squeeze pruning. (b) The solution curve after squeeze pruning.

One example of squeeze pruning that is not optimal at a Steiner node is shown as follows: Suppose the candidate solutions at the right branch are $\alpha_{r1} = (1186ps, 13fF, 12)$, $\alpha_{r2} = (1190ps, 150fF, 12)$ and $\alpha_{r3} = (1243ps, 201fF, 12)$, and the candidate solutions at the left branch are $\alpha_{l1} = (1187ps, 20fF, 10)$ and $\alpha_{l2} = (1200ps, 40fF, 10)$. If squeeze pruning are used after merging point, the candidate solutions after merging are $\alpha_1 = (1186ps, 33fF, 22)$, $\alpha_2 = (1190ps, 190fF, 22)$. If squeeze pruning are used before merging point, the candidate solutions after merging are $\alpha_1 = (1186ps, 33fF, 22)$, $\alpha_2 = (1187ps, 221fF, 22)$. It can be seen that squeeze pruning is sub-optimal at merging point except for the case when the squeeze pruning is only performed after last merging point.

Note that, the pre-buffer and aggressive pre-buffer slack pruning techniques prune the second candidate solution when the slope value between every two neighbouring candidate solutions is smaller than a threshold value. We can treat these two techniques as special cases of squeeze pruning if we assume there is a dummy third candidate solution with the slope to the first candidate solution being that threshold value.

V. LIBRARY LOOKUP (LL)

In van Ginneken style algorithms, the size of buffer library is an important factor. Modern designs often have tens of power levels for buffers and inverters. Sometimes it climbs into the hundreds. From the algorithm analysis, the size of buffer library has the square effect on the running time. In practice, this effect appears to be linear, but it is still a bottleneck when we perform buffer insertion with large buffer libraries. On the other hand, it is essential to have a reasonably sized library to obtain sufficient timing performance. In [15], a buffer library selection algorithm is proposed to prune the big library to get small library and use small library to perform buffer insertion.

During van Ginneken style buffer insertion, every buffer in the library is tried for each candidate solution. If there are n candidate solutions at an internal node before buffer insertion and the library consists of m buffers, then mn tentative solutions are evaluated. For example, in Fig. 4 (a), all eight buffers are considered for all n candidate solutions.

However, many of these candidate solutions are clearly not worth considering (such as small buffers driving large capacitance). But van Ginneken style algorithms generate them anyway and let pruning step eliminates the redundant candidate solutions. Instead, we seek to avoid generating poor candidate solutions in the first place and not even consider adding m buffered candidate solutions for each unbuffered candidate solution. We propose to consider each candidate solution in turn. For each candidate solution with capacitance c_i , we look up the best non-inverting buffer and the best inverting buffer that yield the best delay from two pre-computed tables before optimization. In the example shown in Fig. 4 (b), the capacitance c_i results in selecting buffer B3 and inverter I2 from the non-inverting and inverting buffer tables.

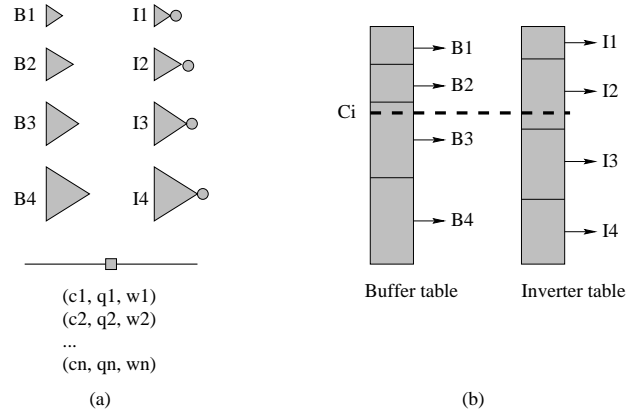


Fig. 4. Library Lookup Example. $B1$ to $B4$ are non-inverting buffers. $I1$ and $I4$ are inverting buffers. (a) van Ginneken style buffer insertion. (b) Library Lookup.

Two pre-computed tables are built as follows: for each possible capacitance value, we give the non-inverting (inverting) buffer with minimum delay when a driver with average size drives this buffer driving this capacitance. Using a table with 300 entries can be quickly computed before buffering and gives more than sufficient granularity.

All $2n$ tentative new buffered candidate solutions can be divided into two groups, where one group includes n candidate solutions with an inverting buffer just inserted and the other group includes n candidate solutions with a non-inverting buffer just inserted. We only choose one candidate solution that yields the maximum slack from each group and finally only two candidate solutions are inserted into the original candidate solution lists. Since the number of tentative new buffered solutions is reduced from mn to $2n$, the speedup is achieved. Also, since only two new candidate solutions instead of m new candidate solutions are inserted to the candidate solution lists (these new solutions could be pruned later), the number of total candidate solutions are reduced. It is equivalent to the case when the buffer library size is only two, but the buffer type can change depending on the downstream load capacitance and all m buffers in the original library can be used if they are listed in the table. This is the major difference between this technique and library pruning in [15] since after library pruning, only those surviving buffers can be used.

VI. EXPERIMENTAL RESULTS

The proposed techniques are implemented together with the buffer insertion algorithm in C and are tested on a SUN SPARC workstations with 400 MHz and 2 GB memory. We test our speedup techniques on three groups of nets from industry ASICs with 300K+ gates. They have been placed and require physical synthesis to optimize timing and fix electrical violations. The first group are extracted from one ASIC chip and consists 1000 most time consuming nets for the algorithm of Lillis et al. [6]. We named it as ChipA-1K. The second group and third group are extracted from another ASIC chip and consist 1000 and 5000 most time consuming nets for the algorithm in [6], respectively. We named them as ChipB-1K and ChipB-5K. We choose the third group since this group includes more nets of small and middle size. The net information for these three groups is shown in Table I.

The buffer library (denote by Full) consists of 24 buffers, in which 8 are non-inverting buffers and 16 are inverting buffers. The range of driving resistance is from $120\ \Omega$ to $945\ \Omega$, and the input capacitance is from $6.27\ fF$ to $121.56\ fF$. We use a scaled Elmore delay as interconnect delay, and apply the $10ps$ rule to choose the most cost efficient solution from a set of solutions with different cost-slack tradeoff.

Table II shows the simulation results for these three test groups. In each experiment, we show the total slack improvement after buffer insertion, the number of inserted buffers for all nets and the total CPU time of the buffer insertion algorithm with our speedup techniques, APSP, SqP and LL. Finally we show the results when three techniques are combined. For comparison, we also show the results of the algorithm of Lillis et al. (baseline) [6] and pre-buffer slack pruning technique (PSP) [14]. Note that for the SqP and LL, we also combine them with pre-buffer slack pruning technique. For APSP, the resistance is chosen as 5% of the range from minimum buffer resistance to the maximum buffer resistance plus the minimum resistance. In [14], it is claimed that pre-buffer slack pruning can achieve up to 17 times speedup. This huge speedup is achieved mainly because the size of test nets and the number of buffer locations in [14] are much larger than our test cases.

The results show that our speedup techniques can provide $9\times$ to $25\times$ speedup over baseline buffer insertion while the slack only degrades by 2-3%. From the table, for ChipB-1K and ChipB-5K nets, the slack improvement of our speedup techniques is slightly greater than the baseline algorithm in few cases. This is due to the usage of the $10ps$ rule in which the most cost-efficient solution is selected instead of the maximum slack solution.

It is obvious that if the size of buffer library becomes smaller, the running time can also go down. However, the solution quality may also degrade. Since our three techniques are independent of the library size, it is interesting to compare our techniques with the baseline algorithm with different buffer libraries. We adopt the buffer library selection algorithm [15] and generate four different buffer libraries from our original library. The libraries are named Tiny, Small, Medium and Large, and they have 4, 6, 9 and 13 non-inverting and

inverting buffers respectively. Then we run pre-buffer slack pruning technique on each library, and the results for three test groups are shown in the Table III. For the ease of comparison, the results for APSP+SqP+LL on original Full library are also shown in the last row for each test group. From the results we can see that with APSP+SqP+LL, the running time is even faster (up to $3\times$ faster) than the results on Tiny buffer library, while we still achieve better slack improvement (up to 1%) and use less buffers. This shows that applying our techniques on a large buffer library could achieve both better slack improvement and faster running time than applying traditional van Ginneken style algorithm on a small buffer library.

VII. CONCLUSION

In this paper, we show that by applying approximation techniques, the presumably fast van Ginneken's buffer insertion algorithm can be further accelerated even for nets of medium or small size. Three speedup techniques are proposed for the practical scenario of using buffer library and considering cost-slack tradeoff. Extensive experiments on industrial designs show that $9\times$ to $25\times$ speedup is achieved through these techniques with only 2%-3% degradation on slack.

REFERENCES

- [1] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, "Repeater scaling and its impact on cad," *IEEE Trans. CAD*, vol. 23, no. 4, pp. 451–463, 2004.
- [2] P. J. Osler, "Placement driven synthesis case studies on two sets of two chips: Hierarchical and flat," in *ISPD*, 2004, pp. 190–197.
- [3] L. P. P. van Ginneken, "Buffer placement in distributed RC-tree network for minimal elmore delay," in *ISCAS*, 1990, pp. 865–868.
- [4] W. Shi and Z. Li, "A Fast Algorithm for Fast Buffer Insertion," *IEEE Trans. CAD*, to appear.
- [5] W. Shi and Z. Li, "An $O(n\log n)$ time algorithm for optimal buffer insertion," in *DAC*, 2003, pp. 580–585.
- [6] J. Lillis, C. K. Cheng, and T.-T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Trans. Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, 1996.
- [7] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *DAC*, 1999, pp. 479–484.
- [8] C.-P. Chen and N. Menezes, "Noise-aware repeater insertion and wire sizing for on-chip interconnect using hierarchical moment matching," in *DAC*, 1999, pp. 502–506.
- [9] J. Lillis, C. K. Cheng, T. T. Y. Lin, and C. Y. Ho, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," in *DAC*, 1996, pp. 395–400.
- [10] M. Hrkic and J. Lillis, "S-tree: a technique for buffered routing tree synthesis," in *DAC*, 2002, pp. 578–583.
- [11] M. Hrkic and J. Lillis, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages," in *ISPD*, 2002, pp. 98–103.
- [12] T. Okamoto and J. Cong, "Buffered steiner tree construction with wire sizing for interconnect layout optimization," in *ICCAD*, 1996, pp. 44–49.
- [13] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *DAC*, 1998, pp. 362–367.
- [14] W. Shi, Z. Li, and C. J. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *ASPAC*, 2004, pp. 609–614.
- [15] C. J. Alpert, R. G. Gandham, J. L. Neves, and S. T. Quay, "Buffer library selection," in *ICCD*, 2000, pp. 221–226.

TABLE I
NET INFORMATION.

	# sinks ≤ 5	$5 < \# \text{ sinks}$ ≤ 20	$20 < \# \text{ sinks}$ ≤ 50	$50 < \# \text{ sinks}$ ≤ 100	# sinks > 100
# nets in ChipA-1k	944	56	0	0	0
# nets in ChipB-1k	0	29	581	345	45
# nets in ChipB-5k	2	1478	2956	513	51

TABLE II

SIMULATION RESULTS FOR CHIPA-1K, CHIPB-1K AND CHIPB-5K NETS ON FULL LIBRARY CONSISTING OF 24 BUFFERS. BASELINE ARE THE RESULTS OF THE ALGORITHM OF LILLIS ET AL. [6]. PSP ARE RESULTS OF PRE-BUFFER SLACK PRUNING TECHNIQUE [14].

Test Case	Algorithm	Slack Imp. (ns)	# of Buffers	CPU (s)	speedup
ChipA-1K	Baseline	5954.93	9895	315.14	1
	PSP	5954.93	9895	280.67	1.12
	APSP	5954.84 (-0.001%)	9893	267.33	1.18
	PSP+SqP	5954.91 (-0.000%)	9895	185.56	1.70
	PSP+LL	5945.47 (-0.16%)	9723	43.87	7.18
	APSP+SqP+LL	5945.44 (-0.16%)	9724	33.50	9.41
ChipB-1K	Baseline	1310.62	9295	1861.26	1
	PSP	1310.62	9295	860.32	2.16
	APSP	1311.64 (+0.08%)	9475	737.38	2.52
	PSP+SqP	1311.04 (+0.03%)	9482	433.66	4.29
	PSP+LL	1288.29 (-1.7%)	9370	144.89	12.85
	APSP+SqP+LL	1290.01 (-1.57%)	9746	75.00	24.82
ChipB-5K	Baseline	2868.86	30438	3577.38	1
	PSP	2868.86	30438	1881.78	1.90
	APSP	2870.32 (+0.05%)	30702	1692.77	2.11
	PSP+SqP	2868.95 (+0.03%)	30822	1036.20	3.45
	PSP+LL	2782.25 (-3.02%)	29080	312.96	11.43
	APSP+SqP+LL	2785.36 (-3.00%)	29776	175.66	20.36

TABLE III

SIMULATION RESULTS FOR CHIPA-1K, CHIPB-1K, CHIPB-5K NETS ON DIFFERENT LIBRARIES. THE NUMBER AFTER EACH LIBRARY IS THE LIBRARY SIZE. PSP ARE RESULTS OF PRE-BUFFER SLACK PRUNING TECHNIQUE [14].

Test Case	Library	Algorithm	Slack Imp. (ns)	# of Buffers	CPU (s)
ChipA-1k	Tiny(4)	PSP	5904.61	9864	59.09
	Small(6)	PSP	5912.07	9843	77.48
	Medium(9)	PSP	5949.30	9720	101.27
	Large(13)	PSP	5951.46	9771	144.32
	Full(24)	APSP+SqP+LL	5945.44	9724	33.50
ChipB-1k	Tiny(4)	PSP	1287.11	10235	220.02
	Small(6)	PSP	1298.06	9511	237.47
	Medium(9)	PSP	1305.26	9274	336.19
	Large(13)	PSP	1306.93	9292	460.69
	Full(24)	APSP+SqP+LL	1290.01	9746	75.00
ChipB-5k	Tiny(4)	PSP	2755.03	32361	460.43
	Small(6)	PSP	2802.99	29758	509.72
	Medium(9)	PSP	2844.57	29895	737.19
	Large(13)	PSP	2853.04	30225	1009.91
	Full(24)	APSP+SqP+LL	2785.36	29776	175.66