Simultaneous Driver Sizing and Buffer Insertion Using a Delay Penalty Estimation Technique

Charles Alpert[1], Chris Chu[2], Gopal Gandham[3], Milos Hrkic[4],
Jiang Hu[5], Chandramouli Kashyap[1], Stephen Quay[1]

1. IBM Corp., Austin, TX 78758
2. Iowa State University, Dept. of Electrical and Computer Engineering, Ames, IA 50011
3. IBM Corp., Hopewell Junction, NY 12533
4. University of Illinois at Chicago, CS Dept., Chicago, IL 60607
5. Texas A&M University, Dept. of Electrical Engineering, College Station, TX 77843

{*calpert, gopalg, vchandra, quayst*}*@us.ibm.com,*
cnchu@iastate.edu, mhrkic@cs.uic.edu, jianghu@ee.tamu.edu

Abstract

To achieve timing closure in a placed design, buffer insertion and driver sizing are two of the most effective transforms that can be applied. Since the driver sizing solution and the buffer insertion solution affect each other, sub-optimal solutions may result if these techniques are applied sequentially instead of simultaneously. We show how to simply extend van Ginneken's buffer insertion algorithm to simultaneously incorporate driver sizing and introduce the idea of a delay penalty to encapsulate the effect of driver sizing on the previous stage. The delay penalty can be pre-computed efficiently via dynamic programming. Experimental results show that using driver sizing with a delay penalty function obtains designs with superior timing and area characteristics.

1 Introduction

Of all the techniques that can be applied in a physical synthesis optimization, driver sizing and buffer insertion are perhaps the two most effective. Typically these operations are performed sequentially, perhaps even iteratively alternating between the two optimizations. The problem is that the two optimizations affect each other which means optimizing them in sequence can yield to a solution that is sub-optimal relative to optimizing them simultaneously.

Consider the example in Figure 1(a) of a woefully underpowered AND gate driving a long interconnect to a single sink. Left alone, a net like this will likely have both prohibitively large delay and poor signal integrity. By applying driver sizing first as in (b), the algorithm will invariably choose an extremely large driver to handle the large capacitive load. Although this will improve the delay characteristics somewhat, the net still will likely need buffers to deal with the resistive interconnect. If one applies buffer insertion first, a solution like that in (c) will result. In this case, buffers are added immediately after the driver, which has the effect of artificially "powering up" the driver before propagating the signal down any significant length of interconnect. The solution in (d) resulting from a simultaneous optimization contains a more reasonably sized driver than that in (b) and uses fewer buffers than in (c). Applying buffer insertion and driver sizing sequentially cannot obtain this solution.

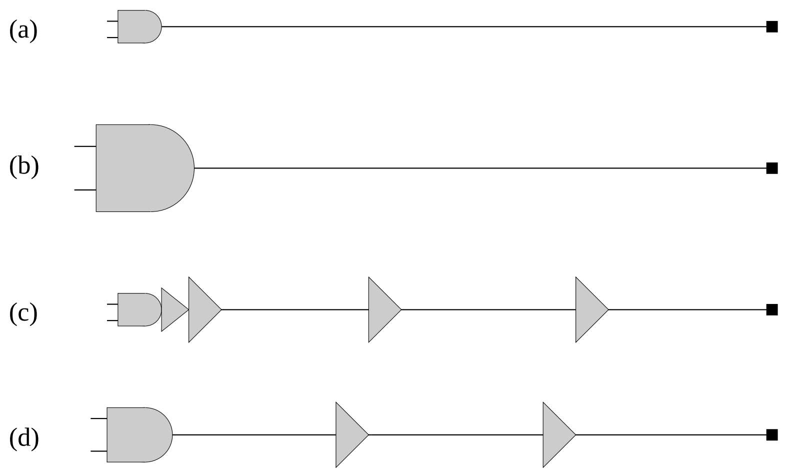

Figure 1: Example single-sink net with long interconnect and (a) an underpowered driver. If one applies (b) driver sizing, then an oversized driver results, but applying (c) buffer insertion leads to additional powering up buffers near the source. The best solution (d) results from simultaneous buffer insertion and driver sizing.

The fundamental approach in buffer insertion is van Ginneken's dynamic programming algorithm [15] which finds the optimal solution for a given Steiner tree and a single buffer type. The reason that this algorithm has become a classic in the field is that its basic approach can be extended to handle many formulations. For example, Lillis *et al.* [9] extends the algorithm to trade off solution quality with buffering resources and use a buffer library with inverters and repeaters. Alpert *et al.* [2] extends the algorithm to simultaneously avoid noise and later in [3] to use higher-order delay models.

If one does not accept the restriction that the Steiner tree is fixed before buffer insertion is performed, then a suite of other heuristics emerge [8] [12] [13] [14] [16]. These heuristics all in some way exploit the dynamic programming paradigm but in a manner that allows multiple tree topologies to be considered. Several works have also proposed simultaneous buffer insertion and wire sizing optimization. The works of [1] and [9] incorporate wire sizing into the van Ginneken framework though several other types of techniques have been proposed (e.g., [5][11]).

One extension that has not yet been proposed (until now) is simultaneous driver sizing. Perhaps this is not surprising since all the previous discussed buffer insertion, Steiner topology, and wire sizing optimizations can be applied to nets independently, while driver sizing cannot. Cong and Koh [4] did propose simultaneous driver and wire sizing, where the driver is modeled as a chain of cascaded drivers.

If one improves the timing characteristics of a given net with these optimizations, then the overall timing is guaranteed not to degrade. However, if one sizes the driver during buffer insertion, the choice of the driver can affect the paths upstream in the timing graph. A local decision to improve the timing for a particular net can result in increased input capacitance on the previous logic stage, thereby degrading the overall timing.

We propose to extend van Ginneken's algorithm to handle driver sizing by treating the source node as a "driver library". We make no assumptions about the relationship between the driver size and key parasitics; rather, we simply assume the existence of functions which can make queries on the delay, slew and input capacitance of a particular gate. This allows one to consider driver sizing on gates with different "footprints" as opposed to the typical assumption of a single scaled footprint.

To try to mitigate the effect of the driver choice upstream without having to query the timing graph, we propose a method for adding a "delay penalty" depending on the choice of the driver. We show that this technique yields better area and timing characteristics than either buffer insertion alone or buffer insertion and driver sizing without a delay penalty.

The remainder of the paper is as follows. Section 2 explains how to extend van Ginneken's algorithm to incorporate driver sizing. Section 3 explains the idea behind our delay penalty calculation, while Section 4 presents the algorithm and extensions. We present experimental results in Section 5 and conclude in Section 6.

2 Simultaneous Driver Sizing and Buffer Insertion

Van Ginneken's algorithm starts from the sinks and recursively builds a set of buffer insertion "candidate" solutions until a single set of candidates is built for the source of the given Steiner tree. This set of candidates is completely independent of the driver strength. After the candidates for the source node are computed, the driver delay is taken into account for each candidate, then the candidate which maximizes the minimum slack to each sink is returned as the solution. This procedure is optimal for a given Steiner tree and the Elmore delay model.

Extending van Ginneken's algorithm to handle driver sizing is fairly straightforward. Assume we have a driver library containing various implementations and/or sizings of the same logic function as the original driver. If p is a particular driver type, let $Delay(p, c)$ be the delay through this gate type driving capacitance c, and let C_p be the input capacitance of the driver.

Figure 2 shows pseudocode of this extension. Step 1 computes the set S of all candidates at the source. Step 2 initializes the new set of candidates S' to the empty set. Step 3 iterates through the candidates in S. For each such candidate, Step 4 generates a new candidate for each driver type p and adds it to S'. Assume for now that $\mathcal{D}(C_p) = 0$. Finally, Step 5 returns the solution in S' with maximum slack.

```
Van Ginneken's algorithm with Driver Sizing
1. Set $S$ to list of all candidate solutions at source.
2. Set $S' = \emptyset$.
3. For each candidate $(c, q) \in S$ do
4.    For each driver type $p$ in driver library
        Let $q' = q - Delay(p, c) - \mathcal{D}(C_p)$.
        Set $S' = S' \cup \{(c, q')\}$.
5. Return candidate $(c, q')$ such that
      $q' = \max\{q \mid (c, q) \in S'\}$.
```

Figure 2: Van Ginneken's algorithm with simultaneous driver sizing.

The complexity of the algorithm is now $O(n^2B^2 + nM)$ where n is the number of possible buffer insertion locations, B is the size of the buffer library, and M is the size of the driver library. If M is less than $O(nB^2)$ (which should be the case most often), the complexity is $O(n^2B^2)$ which is the same as the extension of the van Ginneken algorithm in [9].

The obvious problem with this implementation is that it ignores the impact on the previous stage. In practice, we have observed that the largest (i.e., strongest) driver is almost always chosen which has the most detrimental effect on the previous stage and may worsen the overall delay. In addition, using unnecessarily large drivers waste area and power resources.

One could consider actually temporarily committing the buffer insertion candidate solution to the design, trying various power levels, and timing the design in Step 5. Not surprisingly, it becomes prohibitively expensive to make $O(nM)$ critical path queries. We somehow need to capture the effect on increasing capacitance upstream without actually making queries to the timing analyzer.

We propose to add a delay penalty $\mathcal{D}(C_p)$ that is a function of the input capacitance of p. The larger the input capacitance, the larger the penalty there should be upstream. We now discuss our proposed delay penalty function.

3 Delay Penalty Function

Recall that for a driver p, C_p is the capacitive load of the input pin along the most critical path. The associated delay penalty $\mathcal{D}(C_p)$ is defined as the minimum delay to drive the capacitance C_p by a cascaded buffer chain starting with a buffer with smallest input capacitance.

The reason for this definition is that buffer insertion can always be applied to the preceding net in the critical path. Indeed, one possibility is to insert a series of buffers directly in front of p to isolate the potentially large capacitance C_p on the driver of the previous logic stage. We use the minimum delay needed to isolate the capacitance as our estimation of the delay penalty. This estimation is pessimistic because better ways to insert buffers may be possible. However, this should not be a problem since the estimation is used only as a *relative* measure to compare different driver sizes.

There are several previous results on calculating the minimum delay of a cascaded buffer chain [6] [7] [10]. However, these results assume that a library of buffers of continuous and unbounded size is used and that all buffers can be characterized by a single linear equation. In reality, cell libraries contains a finite number of buffers of discrete size. Also, buffers of different architectures are used. They may have very different characteristic and hence cannot be characterized by a single equation.

Figure 3 graphs delay penalties for a real library of discrete buffers and a library constructed by continuous scaling of the minimum-sized buffers. The two functions are significantly different. In our experiments, we have observed that when the delay penalty for continuous library is used, drivers tend to be undersized and the slacks of the resulting circuits are not as good. Hence, the delay penalty for discrete library is preferred. The following section presents an efficient algorithm to compute the delay penalty for any discrete buffer library.

4 Delay Penalty Computation

For now, assume all buffers are non-inverting and that buffer delays are independent of input slew. We consider extensions for input slew and inverting buffers later. Suppose a buffer library B consisting of n buffers $B_1, \ldots, B_n$ is given. Let $Delay(B_i, C_L)$ be the delay for buffer B_i to drive a given load C_L, and let C_{B_i} be the input capacitance of buffer B_i. Assume that the buffers are ordered such that $C_{B_1} \leq C_{B_2} \leq \cdots \leq C_{B_n}$.

We define the delay penalty $\mathcal{D}(C_L)$ to be the minimum delay over all possible chains of buffers from B to drive C_L such that the first buffer is B_1. If $C_L \leq C_{B_1}$, then adding a buffer chain only increases the capacitance seen by the previous driver, so $\mathcal{D}(C_L)$ is defined as zero.

The following lemma allows us to use a simple dynamic programming technique to construct optimal buffer chains. Assume that for each $B_i \in B$, $Delay(B_i, C_L)$ is monotone non-decreasing in C_L.

Lemma 1 *In any optimal buffer chain, the load C_L driven by any buffer B_j must be greater than C_{B_j}.*

Proof: Assume that the load C_L driven by B_j is less than or equal to C_{B_j}. Assume B_i directly precedes B_j in the buffer chain. Since $C_L \leq C_{B_j}$ and delay is monotone decreasing, $Delay(B_i, C_L) \leq Delay(B_i, C_{B_j})$. Since $Delay(B_j, C_L) > 0$, removing B_j from the buffer chain reduces the overall delay, which means the chain is not optimal.

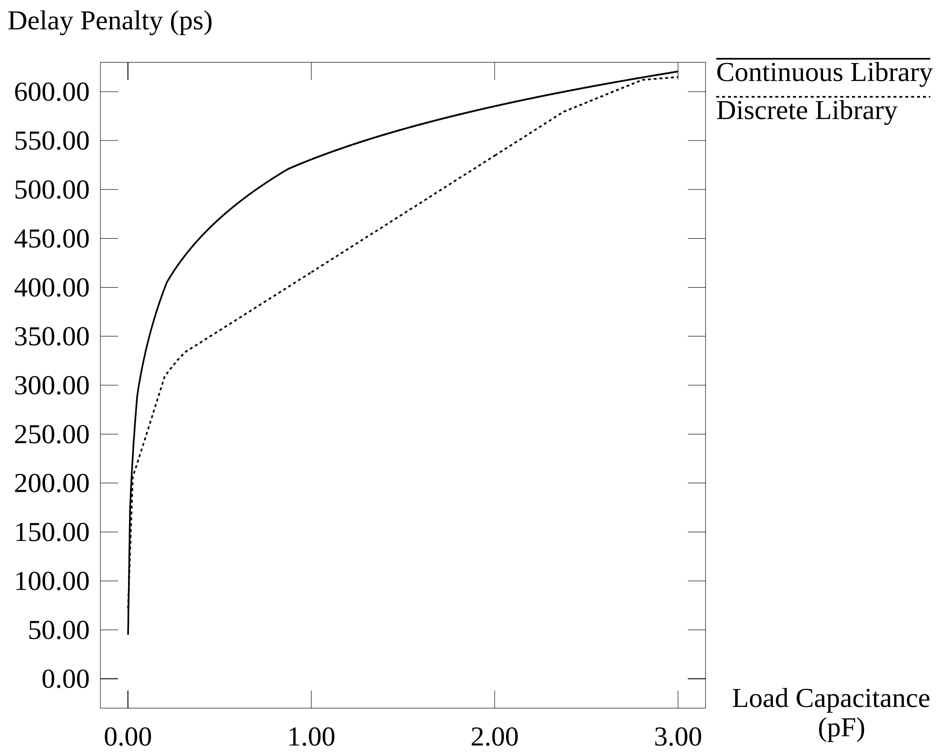

Figure 3: Delay penalty functions for a library of discrete buffers and a library constructed by continuous scaling of the minimum-sized buffer.

The contradiction implies that we must have $C_L > C_{B_j}$. □

We first discuss how to compute delay penalties to drive the input capacitances of buffers (i.e., $\mathcal{D}(C_{B_i})$ for all i). For the optimal buffer chain to drive capacitance C_{B_i}, if B_j is the last buffer in the chain, then according to Lemma 1, $C_{B_j} < C_{B_i}$, which implies $j < i$. That means the optimal buffer chain to drive C_{B_i} can be constructed by appending some buffer B_j to the optimal buffer chain to drive capacitance C_{B_j}, where $j \in \{1, \ldots, i-1\}$. The idea is illustrated in Figure 4. To be more specific, $\mathcal{D}(C_{B_i})$ can be calculated by dynamic programming as follows:

$$\begin{aligned} \mathcal{D}(C_{B_1}) &= 0 \\ \mathcal{D}(C_{B_i}) &= \min_{1 \le j \le i-1} \left[\mathcal{D}(C_{B_j}) + Delay(B_j, C_{B_i})\right] \quad \text{for } i = 2, 3, \ldots, n \end{aligned}$$

For any load with capacitance C_L other than $C_{B_1}, \ldots, C_{B_n}$, the delay penalty is given by finding the buffer B_j such that the delay of the optimal chain through B_j plus the delay of B_j driving C_L is minimized. This is given as:

$$\mathcal{D}(C_L) = \min_{1 \le j \le n} \left[\mathcal{D}(C_{B_j}) + Delay(B_j, C_L)\right]$$

Assume each delay query $Delay(B_i, C)$ takes constant time. Then the time to compute the delay penalty for each buffer in B is $O(|B|^2)$. To compute the delay penalty for a load capacitance for any additional value takes $O(|B|)$ time. Note

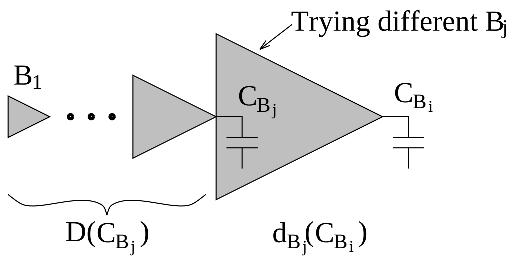

Figure 4: Illustration of the dynamic programming technique to construct optimal buffer chains.

that we only need to compute the delay penalties for driving input capacitances of buffers once for each buffer library. Then the delay penalty and the corresponding optimal buffer chain can be stored for each buffer.

4.1 Slew Consideration

Previously, we had assumed that buffer delay is independent of input slew. We can modify the algorithm to include signal slew in the calculation of delay penalty as follows. First, the buffer chain can be constructed as in Section 4. Then, the slew can be propagated along the buffer chain. Finally, the delay penalty can be computed according to the slew-dependent buffer delay model.

Although this simple extension works well in practice, the buffer chain obtained in Section 4 may not be optimal if slew is considered. The optimal buffer chains with slew consideration can be found by a more elaborate dynamic programming technique. The idea is to propagate both delays and output slews of buffer chains in the dynamic programming algorithm. For each load value, many pairs of delay and output slew, each corresponds to a different buffer chain, are kept as solutions. All the solution pairs are considered when solution pairs for larger load values are constructed.

For buffer B_i, let $Delay(t, B_i, C_L)$ be the delay and $Slew(t, B_i, C_L)$ be the output slew of B_i, where t is the input slew and C_L is the load capacitance. Let t_0 be the input slew to the buffer chain. Let $\mathcal{DT}(t, C_L)$ be the pairs of delay and output slew for a buffer chain to drive C_L if the input slew is t. Then for $i = 2, 3, \ldots, n$, $\mathcal{DT}(t_0, C_{B_i})$ can be calculated by dynamic programming as follows:

$$\mathcal{DT}(t_0, C_{B_i}) \quad = \quad \left\{(d + Delay(t, B_j, C_{B_i}), Slew(t, B_j, C_{B_i})) : (d, t) \in \mathcal{DT}(t_0, C_{B_j}), 1 \le j \le i-1\right\}$$

Note that for any fixed t_0 and C_{B_i}, if there are two pairs in $\mathcal{DT}(t_0, C_{B_i})$ such that one is less than the other in both delay and output slew, then the second pair can be pruned without affecting the optimality of the algorithm.

For any load with capacitance C_L other than $C_{B_1}, \ldots, C_{B_n}$, $\mathcal{DT}(t_0, C_L)$ is given by:

$$\mathcal{DT}(t_0, C_L) \quad = \quad \left\{(d + Delay(t, B_j, C_L), Slew(t, B_j, C_L)) : (d, t) \in \mathcal{DT}(t_0, C_{B_j}), 1 \le j \le n\right\}$$

The delay penalty for a load with capacitance C_L is given by the minimum delay over all the pairs in $\mathcal{DT}(t_0, C_L)$.

4.2 Handling Inverters

Inverters in the buffer library can be handled by a similar idea as in [9]. In additional to $\mathcal{D}(C_{B_i})$, we can define $\mathcal{D}'(C_{B_i})$ similarly for solutions with an inverted output signal. Then $\mathcal{D}(C_{B_i})$ and $\mathcal{D}'(C_{B_i})$ for all i can still be calculated by dynamic programming in $O(|B|^2)$ time as follows:

$$\mathcal{D}(C_{B_1}) = \begin{cases} 0 & \text{if } B_1 \text{ is non-inverting} \\ \infty & \text{if } B_1 \text{ is inverting} \end{cases}$$

$$\mathcal{D}'(C_{B_1}) = \begin{cases} \infty & \text{if } B_1 \text{ is non-inverting} \\ 0 & \text{if } B_1 \text{ is inverting} \end{cases}$$

$$\mathcal{D}(C_{B_i}) = \min_{1\le j\le i-1} \left[\begin{cases} \mathcal{D}(C_{B_j}) + Delay(B_j, C_{B_i}) & \text{if } B_j \text{ is non-inverting} \\ \mathcal{D}'(C_{B_j}) + Delay(B_j, C_{B_i}) & \text{if } B_j \text{ is inverting} \end{cases} \right] \quad \text{for } i = 2, 3, \ldots, n$$

$$\mathcal{D}'(C_{B_i}) = \min_{1\le j\le i-1} \left[\begin{cases} \mathcal{D}(C_{B_j}) + d_{B_j}(C_{B_i}) & \text{if } B_j \text{ is inverting} \\ \mathcal{D}'(C_{B_j}) + d_{B_j}(C_{B_i}) & \text{if } B_j \text{ is non-inverting} \end{cases} \right] \quad \text{for } i = 2, 3, \ldots, n$$

The delay penalty $\mathcal{D}(C_L)$ for each other C_L value can be calculated in $O(|B|)$ time:

$$\mathcal{D}(C_L) = \min_{1\le j\le n} \left[\begin{cases} \mathcal{D}(C_{B_j}) + Delay(B_j, C_L) & \text{if } B_j \text{ is non-inverting} \\ \mathcal{D}'(C_{B_j}) + Delay(B_j, C_L) & \text{if } B_j \text{ is inverting} \end{cases} \right]$$

4.3 Driver and Buffer Area Consideration

Besides causing more delay in preceding stage, a larger driver also occupies more area and potentially induces more buffers in preceding stage. These extra costs associated with driver sizing can be modeled as follows:

$$\text{Total penalty} = \text{Delay penalty} + \alpha * \text{Driver area.}$$

The user-defined constant α converts the driver area into units of time so that it can be added to the delay penalty and to specify the relative importance of delay and area.

4.4 Runtime Reduction by Table-Lookup

For the delay penalty computation technique described in Section 4, each query of delay penalty takes $O(|B|)$ time. A query needs to be made for each candidate buffer insertion solution generated and for each driver size. Hence, the delay penalty computation can be expensive. For example, for the ckt4 in our experiments, buffer insertion and driver sizing are considered for 3000 nets. The library used consists of 48 buffers which requires a total of 6.2 millions queries. As a result, the algorithm is slowed down by about 35%.

To reduce the time spent on delay penalty computation, a table can be constructed to store delay penalty values for a large range of capacitance values before van Ginneken's algorithm is ever called. Since the delay penalty $D(C_L)$ increases faster when C_L is small and slower when C_L is large, a non-uniform interpolation for the lookup table is more efficient. The following function to convert the capacitive load C_L into the index of the table works well in practice:

$$\text{Index in table} = \text{Round-to-Integer}\left(\frac{\log(C_L/C_{min})}{\log(C_{max}/C_{min})} M\right)$$

where M is the number of entries in the table, and C_{min} and C_{max} are a lower bound and an upper bound on C_L, respectively. When $M = 3000$ and $C_{max} = 20000C_{min}$, we observe that this table-lookup method causes less than 0.1% error in delay penalty values and introduces virtually no extra runtime.

5 Experimental Results

Four different driver sizing techniques are incorporated into the van Ginneken style buffer insertion algorithm. They are called ’No’, ’Max’, ’DP’, ’DAP’ below:

- ’No’ – No sizing is done, just like in the original van Ginneken algorithm.
- ’Max’ – The driver size is chosen to optimize the buffered net locally, i.e., the delay penalty is set to zero. This typically leads to choosing one of the strongest available drivers, hence the ’Max’ name.
- ’DP’ – The delay penalty approach is used.
- ’DAP’ – The driver area penalty is added to ’DP’ as explained in Section 4.3. The area constant α is set to $1ps/\mu m^2$.

These four algorithms are applied to five industry circuit designs. For each circuit, we first determine a subset of nets to optimize based on net capacitances and criticalities. Then the integrated buffer insertion and driver sizing algorithms are applied to each net in the list sequentially. The results are summarized by the following measurements in Table 1:

- **Nets seen** is the number of nets for which buffer insertion and driver sizing is considered. The number varies slightly between the algorithms due to newly created nets. For example, one might try to optimize a net created by a previous buffer insertion solution.
- **Buffering** presents the total number of buffers inserted and the total area occupied by the buffers.
- **Driver Sizing** presents the number of drivers that were sized up to large size, the number down to a smaller size, and the total change in area from the sizing.
- **Area Δ** gives both the total and percentage change in area resulting from applying the optimization algorithm.
- **Negative Slack** gives the slack of the most timing critical path and the total number of nets remaining with negative slack values.

We make the following observations:

- The ’No’ algorithm, corresponding to no driver sizing inserts more buffers than the other three schemes. It also yields the most negative paths remaining for four of the five circuits, though the worst slack path is often competitive with the other approaches.
- The ’Max’ algorithm sizes up the most drivers by far (though it also sizes down several) and also consumes the most area. This is especially prevalent for ckt4 where its sizing consumes 25.7% of the total circuit area. Using this much area will certainly cause both problems for power and the placement legalization program.
- ’DP’ uses significantly less total area than ’Max’ and slightly less than ’No’ overall. It also generates better performance in terms of the most critical path and the number of negative slack nets.
- The ’DAP’ algorithm obtains similar results to ’DP’ (though arguably not quite as strong), but with significant area savings.

Overall, the delay penalty function appears to work as intended. It prevents the significant gross oversizing done when there is no delay penalty while actually improving the overall timing of the design.

6 Conclusions

We have shown how to extend van Ginneken's classic buffer insertion algorithm to simultaneously perform driver sizing without increasing time complexity. A key component to our approach is our ability to handle the effect of increasing the capacitance on the previous net without actually querying the timing graph. This is accomplished using an efficient dynamic programming approach to build optimal discrete buffer chains. Our experiments show that the delay penalty function effectively reduces total area and also obtains better overall timing. In future work, we plan to study the impact of this approach within a physical synthesis environment.

References

[1] C. J. Alpert, A. Devgan, J. P. Fishburn, and S. T. Quay, "Interconnect Synthesis Without Wire Tapering," *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 90–104, Jan. 2001.

[2] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion for noise and delay optimization," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 362–367, 1998.

[3] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 479–484, 1999.

[4] J. Cong and C.-K. Koh, "Simultaneous Driver and Wire Sizing for Performance and Power Optimization", *IEEE Trans. VLSI Systems*, 2(4), pp. 408-425, December 1994.

[5] J. Cong, C.-K. Koh, and K. S. Leung, "Simultaneous buffer and wire sizing for performance and power optimization," in *Proc. Int. Symp. Low-Power Electronic Design*, pp. 271–276, August 1996.

[6] N. Hedenstierna and K. O. Jeppson, "CMOS circuit speed and buffer optimization," *IEEE Trans. Computer-Aided Design*, CAD-6(2):270–281, March 1987.

[7] R. C. Jaeger, "Comments on 'An optimized output stage for MOS integrated circuits'," *IEEE J. Solid-State Circuits*, SC-10(3):185–186, June 1975.

[8] J. Lillis, C. K. Cheng, and T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proceedings of the Great Lake Symposium on VLSI*, pp. 148–153, 1996.

[9] J. Lillis, C. K. Cheng, and T. Y. Lin, "Optimal wire sizing and buffer insertion for low and a generalized delay model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 437–447, Mar. 1996.

[10] H. C. Lin and L. W. Linholm, " An optimized output stage for MOS integrated circuits," *IEEE J. Solid-State Circuits*, SC-10(2):106–109, April 1975.

[11] Yu-Yen Mo and Chris Chu, "A hybrid dynamic/quadratic programming algorithm for interconnect tree optimization," *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 680–686, May 2001.

[12] T. Okamoto and J. Cong, "Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion," in *ACM Physical Design Workshop*, pp. 1–6, 1996.

[13] A. H. Salek, J. Lou, and M. Pedram, "A simultaneous routing tree construction and fanout optimization algorithm," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 625–630, 1998.

[14] A. H. Salek, J. Lou, and M. Pedram, "MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 472–478, 1999.

[15] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 865–868, 1990.

[16] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 96–99, 1999.

ckt1: 73K cells										
	Nets	**Buffering**		**Driver Sizing**			**Area Δ**		**Negative Slack**	
	seen	**insert**	**area**	**up**	**down**	**area Δ**	**Total**	**%**	**worst**	**nets**
No	15607	2120	615037	0	0	0	615037	7.8	-2141.90	13104
Max	15331	866	210882	12288	150	2552278	2763159	34.9	-2167.97	12459
DP	14706	658	140522	2796	151	351624	492147	6.2	-1727.57	11567
DAP	14791	646	136629	2387	147	280629	417257	5.3	-1772.72	11738

ckt2: 93K cells										
	Nets	**Buffering**		**Driver Sizing**			**Area Δ**		**Negative Slack**	
	seen	**insert**	**area**	**up**	**down**	**area Δ**	**Total**	**%**	**worst**	**nets**
No	491	349	143780	0	0	0	143780	1.1	-710.36	383
Max	484	137	57722	312	6	48245	105967	0.8	-431.43	170
DP	485	147	58212	236	5	34751	92963	0.7	-432.39	200
DAP	483	142	57551	216	7	30906	88457	0.7	-536.03	236

ckt3: 196K cells										
	Nets	**Buffering**		**Driver Sizing**			**Area Δ**		**Negative Slack**	
	seen	**insert**	**area**	**up**	**down**	**area Δ**	**Total**	**%**	**worst**	**nets**
No	28395	3697	1226767	0	0	0	1226767	0.8	-6413.65	83726
Max	28225	2833	883004	19419	531	3742457	4625461	3.0	-7373.20	85861
DP	28204	2919	868776	3478	624	357624	1226400	0.8	-6756.68	82353
DAP	28220	2978	880873	2312	671	158253	1039127	0.7	-6335.87	82346

ckt4: 285K cells										
	Nets	**Buffering**		**Driver Sizing**			**Area Δ**		**Negative Slack**	
	seen	**insert**	**area**	**up**	**down**	**area Δ**	**Total**	**%**	**worst**	**nets**
No	29907	1942	4645515	0	0	0	4645515	6.4	-2253.69	147632
Max	29199	1076	2749371	16670	431	15814341	18563712	25.7	-2539.32	144666
DP	29272	1059	2441902	5599	881	3341002	5782904	8.0	-2254.90	143162
DAP	29842	1243	3039484	1408	649	244240	3283723	4.5	-2353.36	146505

ckt5: 303K cells										
	Nets	**Buffering**		**Driver Sizing**			**Area Δ**		**Negative Slack**	
	seen	**insert**	**area**	**up**	**down**	**area Δ**	**Total**	**%**	**Worst**	**Nets**
No	22033	9444	1820760	0	0	0	1820760	0.7	-4127.03	66484
Max	21172	8621	1550088	12576	946	898553	2448641	0.9	-4127.03	64837
DP	21448	7921	1435692	5741	628	207430	1643122	0.6	-4127.03	64613
DAP	21579	8149	1481136	4149	745	135158	1616294	0.6	-4127.03	65013

Table 1: Comparison of four different driver sizing techniques on five industry circuits.