Fast Buffer Insertion for Yield Optimization Under Process Variations*

Ruiming Chen and Hai Zhou
Electrical Engineering and Computer Science
Northwestern University, Evanston, IL 60208

ABSTRACT

With the emerging process variations in fabrication, the traditional corner-based timing optimization techniques become prohibitive. Buffer insertion is a very useful technique for timing optimization. In this paper, we propose a buffer insertion algorithm with the consideration of process variations. We use the solutions from the deterministic buffering that sets all the random variables at their nominal values to guide the statistical buffering algorithm. Our algorithm keeps the solution lists short, and always achieves higher yield than the deterministic buffering. The experimental results demonstrate that the exiting approaches cannot handle large cases *efficiently* or *effectively*, while our algorithm handles *large* cases very efficiently, and improves the yield more than 12% on average.

1. INTRODUCTION

With shrinking geometries in deep sub-micron technology, process variation becomes a prominent phenomenon in fabrication. With process variations, the traditional corner-based analysis and optimization techniques become prohibitive. Recently, there emerged many statistical static timing analysis (SSTA) approaches [1, 2], which greatly speed up the analysis by propagating the distributions instead of single values. Based on those, some statistical optimization techniques on gate sizing, buffer insertion [3–6] also emerged. For the buffer insertion problem, Saxena *et al.* [7] predicted synthesis blocks to have 70% of their cell count dedicated to interconnect buffers within a few process generations. With this huge number of buffers, there is an increasing demand to maximize the timing yield under process variations. The timing yield in this paper is defined as the probability that the maximal required arrival time at the root of a RC-tree that can be achieved by buffering satisfies a given timing constraint.

In this paper, we consider the timing yield optimization problem of buffer insertion under process variations. Since the number of non-inferior statistical solutions is very large, a main challenge in the statistical buffering is how to prune the solutions such that the number of solutions and the sacrifice of the yield are both minimized. There are some work that considered the buffering problem under process variations [3–6, 8]. Khandelwal *et al.* [3] considered only the wire length variation, and the pruning approaches are expensive. Davoodi *et al.* [5] considered the correlations between the delay and the downstream capacitance, but the pruning approach is still prohibitive. Both techniques use a two-phase scheme for the merge of solution lists: the solutions are generated in the first phase, and pruned in the second phase. Then for the merge procedure that merges two solution lists with m solutions and n solutions respectively, these techniques run in at least $O(m^2n^2)$ time (the number of solutions before pruning is $O(mn)$, and the prune techniques compute the relation between each pair of solutions), which is prohibitive for large m and n. Xiong *et al.* [4] proposed to compute the joint probability density function (JPDF) of solutions numerically, which is demonstrated to be very inefficient [6]. Xiong and He [6] ignored correlations between the delay and the downstream capacitance, tried to propose a transitive closure technique for probabilistic buffering, and claimed that a direct extension of the deterministic buffering algorithms as in [9, 10] can be used in the statistical situation. But the ordering property[1] is not generally true, and as shown in our experimental result part, the quality of buffering solutions from [6] is not good compared with the deterministic buffering that assumes that all the random variables have the nominal values. Deng *et al.* [8] claimed that the consideration of process variations is not necessary for the buffering of two-pin nets. But the conclusion does not hold for nets with multi pins according to the results in the other existing statistical buffering work.

Table 1: Yield comparison between nominal (Nom) and worst (Wor) scenarios.

	p1	p2	r1	r2	r3	r4	r5
Wor	63.31	73.37	49.00	62.14	69.20	75.62	71.20
Nom	62.88	79.21	62.60	67.03	81.30	78.56	70.65

In this paper, we use the solutions generated by deterministic buffering to guide the statistical pruning. We observed that the deterministic buffering fixing all the random variables at their nominal values (μ) achieves higher yield than the buffering using the worst values ($\mu + 3\sigma$). We use the van Ginneken's algorithm [9] to compute the solutions for these two scenarios respectively. The solution with the minimal delay in the solution list at the root is selected, and the yield of this solution is computed by Monte Carlo simulation. Table 1 shows the comparison results between these two scenarios. All these nets have more than two pins. The results indicate that the solutions from the nominal scenario

*This work was partially supported by NSF under CCR-0238484 and a grant from Intel.

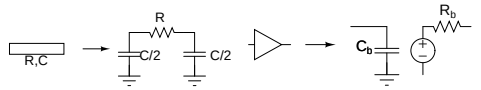
[1]For any two random variable A and B with Gaussian distributions, either $P_r(A \geq B) \geq \gamma$ or $P_r(B \geq A) \geq \gamma$, where γ is a given constant number in (0.5, 1].

are much better than those from the worst scenario for most cases. For example, the solution in nominal scenario gains 13.60% more yield for case "r1". This result is reasonable because since all the random variables are assumed to have Gaussian distributions, the buffering in the worst scenario tends to optimize the costs for the extreme cases, while the buffering in the nominal scenario optimizes the costs for the cases that have much greater probabilities to occur in the samplings. This is also consistent with the analysis in [8], where it is proved that the buffering in the nominal scenario achieves good yield for two-pin nets.

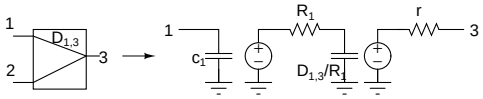
During the merge of the solution lists, we select at most two times the number of the solutions in the deterministic buffering for each solution list, so our algorithm is very efficient. In this paper, we use Elmore delay model to compute the delays of wires and gates, but our buffering framework can also be used when more complex model is used if the following two conditions are satisfied. The first condition is that the nominal samples have large probabilities to occur in the samplings. If this is not the case, we need to select another scenario where the samples have the biggest probabilities to occur as the base of our approach. If the random variables are approximated to have Gaussian distribution, this condition is always satisfied. The second condition is that van Ginneken's algorithm can be used with that model in deterministic situation. This is satisfied by most of the current delay models, since the variants of the van Ginneken's algorithm are still widely used in the industry.

The rest of this paper is organized as follows. Section 2 briefly reviews the existing buffer insertion algorithms for the deterministic scenario. Section 3 presents our buffering algorithm with the consideration of the process variations. In Section 4, the experimental results demonstrate that our algorithm is very efficient, and achieves more than 12% improvement on the yield on average. Finally, the conclusions are drawn in Section 5.

2. PRELIMINARY



(a) Wire model (b) Buffer model



(c) Gate model (from pin 1 to pin 3)

Figure 1: Delay models of wires, buffers and gates.

For simplicity, we use the Elmore delay model to compute the delays of wires, modules and buffers as shown in Fig. 1. Given a routing tree as a distributed RC network, the classic van Ginneken's algorithm [9] for buffer insertion computes non-inferior solutions bottom-up from the sinks to the root. The objective is to insert buffers such that the maximal delay D_{root} from the root to sinks is minimized. Each solution (D_v, C_v) at a node v represents a buffering of the subtree rooted at v having D_v as the maximal delay to the sinks and C_v as the loading capacitance. When a tree at u is composed of a wire (u, v) and a subtree at v, its solution (D_u, C_u) can be computed as follows.

$$D_u = D_v + r(u,v)(C_v + c(u,v)/2), \quad C_u = C_v + c(u,v),$$

where $r(u, v)$ and $c(u, v)$ are the resistance and capacitance of wire (u, v), respectively. When a buffer is inserted at the node v, a new solution (D'_v, C'_v) can be computed similarly:

$$D'_v = D_v + d_b + r_b C_v, \quad C'_v = c_b,$$

where d_b, r_b and c_b are the internal delay, output resistance and gate capacitance of the buffer, respectively. When a node has two branches, assuming that (D_m, C_m) is a solution in one branch and (D_n, C_n) in the other, the combined solution is given as follows.

$$D = \max(D_m, D_n), \quad C = C_m + C_n.$$

The optimal structure of dynamic programming requires that there are no solutions (D_1, C_1) and (D_2, C_2) such that $D_1 \leq D_2$ and $C_1 \leq C_2$ for the same subtree.

The van Ginneken's algorithm traverses the routing tree from the sinks to the root, and at each vertex, it computes the solutions according to the above description, and prunes the inferior solutions using the optimality condition.

3. CORRELATION-AWARE BUFFERING

With the consideration of the process variations, the delay and capacitance parts of each solution become random variables. Here, we assume that all the random variables ($r(u, v)$, $c(u, v)$, d_b, r_b, c_b, D_v and C_v) have the Gaussian distribution. Each random variable X is represented in a canonical first-order form:

$$x_0 + \sum_{i=1}^{m} x_i \epsilon_i + x_{m+1} R_x,$$

where ϵ_i's are independent random variables with the standard Gaussian distribution, x_0 is the mean value of X, x_i's are coefficients, and R_x is an independent random variable with the standard Gaussian distribution. Here, R_x represents the independent variation. This canonical form is available through principal component analysis (PCA) [1, 11].

When a wire $w(u, v)$ is attached to a node v, the computation of the maximal delay involves the multiplication between the random variables ($r(u, v)C_v$ and $r(u, v)c(u, v)$). Keeping all the random variables in a first-order form as

$$x_0 + \sum_{i=1}^{m} x_i \epsilon_i,$$

Xiong and He [6] used the moment-matching approach to compute the multiplication between the Gaussian random variables. If many random variables have their independent variations, the number of non-zero coefficients may become larger and larger during the computation. Thus, we prefer the canonical form that models independent randomness. We need to keep that canonical form during the buffering.

Let

$$C_v = C_{v0} + \sum_{i=1}^{m} \alpha_i \epsilon_i + \alpha_{m+1} R_{C_v},$$

$$D_v = D_{v0} + \sum_{i=1}^{m} \beta_i \epsilon_i + \beta_{m+1} R_{D_v}.$$

Also suppose

$$r(u,v) = r_0 + \sum_{i=1}^{m} \gamma_i \epsilon_i + \gamma_{m+1} R_r,$$

$$c(u,v) = c_0 + \sum_{i=1}^{m} \zeta_i \epsilon_i + \zeta_{m+1} R_c.$$

Then the solution after attaching a wire (u,v) to v has

$$C_u = C_{v0} + c_0 + (\alpha^T + \zeta^T)\Upsilon + \alpha_{m+1} R_{C_v} + \zeta_{m+1} R_c \quad (1)$$

where Υ represents the column vector $(\epsilon_1, ..\epsilon_m)^T$ (α, β, γ and ζ also represent the corresponding column vectors), For C_u, we get

$$\sigma^2(C_u) = \sum_{i=1}^{m} (\alpha_i + \zeta_i)^2 + \alpha_{m+1}^2 + \zeta_{m+1}^2.$$

By moment matching, we get

$$C_u = (C_{v0} + c_0) + (\alpha^T + \zeta^T)\Upsilon + \sqrt{\alpha_{m+1}^2 + \zeta_{m+1}^2} R_{C_u},$$

where R_{C_u} is an independent random variable representing the local variance. For D_u,

$$D_u = K + P\Upsilon + \Upsilon^T Q \Upsilon + R,$$

where

$$\begin{aligned}
K &= D_{v0} + r_0 C_{v0} + r_0 c_0/2 && (2)\\
P &= \beta^T + C_{v0}\gamma^T + r_0\alpha^T + 0.5 c_0 \gamma^T + 0.5 r_0 \zeta^T && (3)\\
Q &= \gamma\alpha^T + 0.5\gamma\zeta^T && (4)\\
R &= r_0\alpha_{m+1}R_{C_v} + C_{v0}\gamma_{m+1}R_r + \alpha^T\Upsilon\gamma_{m+1}R_r \\
&\quad + \gamma^T\Upsilon\alpha_{m+1}R_{C_v} + \gamma_{m+1}R_r\alpha_{m+1}R_{C_v} \\
&\quad + 0.5(r_0\zeta_{m+1}R_c + c_0\gamma_{m+1}R_r + \zeta^T\Upsilon\gamma_{m+1}R_r \\
&\quad + \gamma^T\Upsilon\zeta_{m+1}R_c + \gamma_{m+1}\zeta_{m+1}R_r R_c) + \beta_{m+1}R_{D_v} && (5)
\end{aligned}$$

Then

$$\begin{aligned}
E(D_u) &= K + tr(Q). && (6)\\
E(D_u^2) &= K^2 + PP^T + 2tr(Q^2) + tr(Q)^2 \\
&\quad + \beta_{m+1}^2 + \gamma_{m+1}^2 tr(\alpha\alpha^T) + \alpha_{m+1}^2 tr(\gamma\gamma^T) \\
&\quad + C_{v0}^2\gamma_{m+1}^2 + \gamma_{m+1}^2\alpha_{m+1}^2 + r_0^2\alpha_{m+1}^2 \\
&\quad + 0.25 r_0^2\zeta_{m+1}^2 + 0.25\gamma_{m+1}^2\zeta_{m+1}^2 \\
&\quad + 0.25\zeta_{m+1}^2 tr(\gamma\gamma^T) + 0.25 c_0^2\gamma_{m+1}^2 \\
&\quad + 0.25\gamma_{m+1}^2 tr(\zeta\zeta^T) + \gamma_{m+1}^2 tr(\zeta\alpha^T) \\
&\quad + C_{v0}c_0\gamma_{m+1}^2 + 2Ktr(Q). && (7)
\end{aligned}$$

Thus, we can compute $\sigma^2(D_u)$ according to

$$\sigma^2(D_u) = E(D_u^2) - E(D_u)^2. \quad (8)$$

And

$$\begin{aligned}
cov(D_u, \epsilon_i) &= E(D_u\epsilon_i) \\
&= P_i. && (9)
\end{aligned}$$

Similar as in [2],

$$D_u = (K + tr(Q)) + \sum_{i=1}^{m} P_i\epsilon_i + MR_{D_u} \quad (10)$$

where $M = \sqrt{\sigma^2(D_u) - PP^T}$.

Suppose

$$r_b = r_{b0} + \sum_{i=1}^{m} \xi_i\epsilon_i + \xi_{m+1}R_{rb},$$

$$c_b = c_{b0} + \sum_{i=1}^{m} \theta_i\epsilon_i + \theta_{m+1}R_{cb},$$

$$d_b = d_{b0} + \sum_{i=1}^{m} \lambda_i\epsilon_i + \lambda_{m+1}R_{db}.$$

When a buffer is attached to node v, the new solution has

$$C_v' = c_{b0} + \sum_{i=1}^{m} \theta_i\epsilon_i + \theta_{m+1}R_{cb}, \quad (11)$$

$$D_v' = L + J\Upsilon + \Upsilon^T\xi\alpha^T\Upsilon + H \quad (12)$$

where

$$\begin{aligned}
J &= \beta^T + \lambda^T + r_{b0}\alpha^T + C_{v0}\xi^T && (13)\\
L &= D_{v0} + d_{b0} + r_{b0}C_{v0} && (14)\\
H &= \beta_{m+1}R_{D_v} + \lambda_{m+1}R_{db} + r_{b0}\alpha_{m+1}R_{C_v} \\
&\quad + \xi^T\Upsilon\alpha_{m+1}R_{C_v} + \xi_{m+1}R_{rb}C_{v0} \\
&\quad + \xi_{m+1}R_{rb}\alpha^T\Upsilon + \xi_{m+1}R_{rb}\alpha_{m+1}R_{cv}. && (15)
\end{aligned}$$

Then, similarly, we get

$$\begin{aligned}
E(D_v') &= L + \xi^T\alpha && (16)\\
\sigma^2(D_v') &= JJ^T + \beta_{m+1}^2 + \lambda_{m+1}^2 + r_{b0}^2\alpha_{m+1}^2 \\
&\quad + \alpha_{m+1}^2\xi^T\xi + C_{v0}^2\xi_{m+1}^2 + \xi_{m+1}^2\alpha^T\alpha \\
&\quad + \xi_{m+1}^2\alpha_{m+1}^2 + 2tr((\xi\alpha^T)^2). && (17)\\
cov(D_v', \epsilon_i) &= J_i. && (18)
\end{aligned}$$

Similar as in [2],

$$D_v' = (L + \xi^T\alpha) + \sum_{i=1}^{m} J_i\epsilon_i + NR_{D_v'}, \quad (19)$$

where $N = \sqrt{\sigma^2(D_v') - JJ^T}$.

The merge of solutions involves the "Max" operation between random variables. The approach in [2] is used to handle this.

Now comes our correlation aware buffering algorithm. A main challenge in buffering is how to prune inferior solutions. In deterministic scenario, if $D_1 \le D_2$ and $C_1 \le C_2$, the solution (D_2, C_2) is inferior. In statistical scenario, D and C are random variables, thus, if

$$Pr(D_1 \le D_2, C_1 \le C_2) = 1,$$

(D_2, C_2) is inferior. It can be easily proved that no two random variables with Gaussian distributions satisfy this condition unless one of them is equal to the sum of a constant and the other one. Thus, we need to relax this condition to prune more solutions. It becomes

DEFINITION 1 (PRUNE RULE). *For (D_1, C_1) and (D_2, C_2), if*

$$Pr(D_1 \le D_2, C_1 \le C_2) \ge \eta,$$

where η is a given probability, (D_2, C_2) is pruned.

We use the approach in [12] to evaluate the bivariate probabilities in this prune rule.

```
Algorithm FSBI(u)
    ▷ Phase 1: deterministic buffering
 1  if u is the root
 2     then
 3          VanGinneken(u)
    ▷ Phase 2: statistical buffering
 4  List lst ← φ
 5  if u is a sink (with delay D_u and capacitance C_u)
 6     then
 7          return {(D_u, C_u)}
 8  for each child node v of u
 9        do lst_v ← FSBI(v)
10          for each solution (D_v, C_v) in S
11               do
12                    update the (D_v, C_v) by attaching
                        the wire (u, v)
13          if u is a buffering location
14             then
15                    find the solution with the minimal
                        mean value of D among the new
                        solutions generated by attaching
                        a buffer near u
16                    insert this solution into lst_v in the
                        increasing order of the mean value
                        of C
17          prune lst_v according to the prune rule
18  lst ← merge all the lst_v's
19  return lst
```

Figure 2: FSBI algorithm

The general flow of our algorithm, called FSBI, is shown in Fig. 2. The input of this algorithm is the root of the routing tree. Without loss of the generality, we assume that there is only one type of buffer in the library. FSBI has two stages: the first stage is the deterministic buffering, and the second is the statistical buffering.

In the first stage, all the random variables have the nominal values, and FSBI uses the van Ginneken's algorithm to compute the non-inferior solution list at each node.

In the second stage, all the random variables have their distributions, and the solutions in a solution list are sorted in the increasing order of the mean value of the downstream capacitance (C). When a buffer is attached to a node, we only generate *one* solution: the solution with the minimal mean delay. This can keep the number of solutions small,

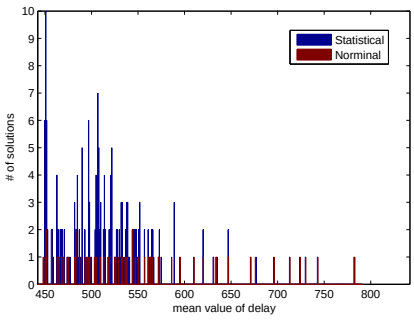


Figure 3: The distributions of the mean delays of the solutions in different scenarios for "r2".

and does not greatly affect the yield in practice.

The merge of two solution lists with m solutions and n solutions respectively generates $O(mn)$ solutions, so if the number of merge operations is large, the algorithm becomes prohibitive. As shown in Fig. 3, even after the pruning with the previously introduced rule, the number of statistical solutions is still too large. We observed that the downstream capacitance computation in the van Ginneken's algorithm does not involve any non-linear operations (actually all the operations are additions), thus the mean value of the downstream capacitance is always a linear function of the mean value of the operands. For example, if a wire (u, v) is attached to node u,

$$\mu_{C_u} = \mu_{C_v} + \mu_{c(u,v)}.$$

Thus, for each deterministic solution, we can always find statistical solutions that have the same mean C values in the statistical solution space. Now suppose (D', C') is the statistical representation of a deterministic solution (D, C), then we have $\mu(C') = C$. In addition, the D of a solution depends on the C of the downstream solutions, while the C does not depend on the D of the downstream solutions. Therefore, for each deterministic solution (D, C), if we always pick one statistical solution (D'', C'') that satisfies

$$(\mu(D'') < \mu(D')) \wedge (\mu(C'') \leq \mu(C') = C),$$

the yield is expected to be higher than the yield computed by the deterministic buffering in nominal scenario. The number of statistical solutions is reduced to the number of deterministic solutions, so the algorithm is efficient.

We use a one-phase merge procedure in FSBI. The pseudocode of the merge procedure is shown in Fig. 4. Without loss of generality, we assume that each node has at most two branches. In the first stage of FSBI, the deterministic solutions from the left branch and the right branch are put into the $leftlist$ and $rightlist$ fields of each node, respectively, in the increasing order of C. The input of the MERGE is the current node t, and the statistical solution lists $llst$ and $rlst$ from t's branches. First, the MERGE calls the function COMPUTEDENSITY, which for each interval composed by the C parts of two consecutive deterministic solutions in the recorded solution list, computes the number of statistical solutions whose C's have the mean values located in the interval. This can be done in linear time. COMPUTEDENSITY returns an array storing the number of solutions in each interval.

In order to improve the yield, we need to keep many statistical solutions at the beginning such that we can find a solution that has a small mean D value in each interval during the merge of big solution lists. Thus, the MERGE checks if the number of solutions is much larger than the number of solutions in the deterministic solution lists. If not, we do not need to further prune any solutions, otherwise, we pick one statistical solution from each interval. The subroutine PICKONESOLUTION(LST, COUNT) accomplishes this. It picks the solution having the minimal mean value of D in the current interval, and returns the distance from the last picked solution to current picked solution. In summary, the MERGE sub-routine keeps all the non-inferior solutions when the size of the current statistical solution list is comparable with the size of the solution list of the deterministic buffering, or picks only a small number of the solutions when the current solution list becomes much larger. Using this ap-

```
Algorithm MERGE(t, llst, rlst)
    ▷ lst is a list in the increasing order of the
      mean value of C
 1  lst ← φ
 2  count1 ← COMPUTEDENSITY(t.leftlist, llst)
 3  count2 ← COMPUTEDENSITY(t.rightlist, rlst)
 4  i ← 0
 5  while ( i < length(llst) )
 6        do
 7           if length(llst) ≤ 2 × length(t.leftlist)
 8              then
 9                   sol1 ← llst[i]
10                   i ← i + 1
11              else
12                   sol1 ← llst[i]
13                   p ← PICKONESOLUTION(llst, count1)
14                   i ← i + p
15           j ← 0
16           while ( j < length(rlst) )
17                 do
18                    if length(rlst) ≤ 2 × length(t.rightlist)
19                       then
20                            sol2 ← rlst[j]
21                            j ← j + 1
22                       else
23                            sol2 ← rlst[j]
24                            p ← PICKONESOLUTION(rlst,
                                count2)
25                            j ← j + p
26                    (max(sol1.d, sol2.d), sol1.c + sol2.c) is
                        inserted into the lst
27  return lst
```

Figure 4: Merge sub-routine

proach, we have $\mu(D'') \leq \mu(D')$ and $\mu(C'') \leq C$. Thus, the yield is expected to be at least not worse.

The insertion of one merged solution into the list also checks if this solution is inferior or there exists a solution in the list that is inferior to this solution according to the prune rule. After this checking, if this solution is not inferior, we insert the solution into the list in the increasing order of the mean value of C. Actually, many of the solutions are inferior in practice, so the size of the solution list is always much less than mn. Therefore, this simultaneous merge and prune procedure also greatly improves the efficiency.

4. EXPERIMENTAL RESULTS

The FSBI algorithm is implemented in C++. We use the heuristic II that is claimed to be the best in the three heuristics in [3], and the approach in [6] for comparison. [5] also uses an expensive two-phase merge strategy, and the algorithm in [4] is shown to be very slow in [6], so we do not compare them with ours.

FSBI is tested on all the test cases from [13]. The characteristics of these test cases are shown in Table 2. For the nets with small number of sinks (e.g., the data nets), the existing approaches can be used. So we test our approach on only those nets with big number of sinks. Since the original test cases in [13] do not have the statistical information (e.g., deviation, correlation), we generate the statistical information by ourselves. For each random variable representing the d_b, the resistance, or the capacitance, we randomly generate the coefficients of the ϵ_i's in the canonical form, and enforce that each random variable has 10% deviation from its nominal value. Note that although the test cases in [6] are also derived from the test cases in [13], since we cannot get those test cases with statistical information from [6], our test cases are different from those in [6]. $\eta = 0.90$ in the prune rule. All the experiments were run on a Linux PC with 2.4 GHz Xeon CPU and 2.0 GB memory.

Table 2: The characteristics of the test cases

name	# sinks	# nodes	# buffer locations
p1	269	537	268
p2	603	1205	602
r1	267	533	266
r2	598	1195	597
r3	862	1723	861
r4	1903	3805	1902
r5	3101	6201	3100

The heuristic II in [3] completes the merge procedure as follows. First, it merges each pair of solutions from left branch and right branch respectively, and then computes the prune probability between each pair of solutions in the new solution lists. A graph with the vertices representing the solutions and the edges representing the pruning relations between solutions is constructed. Then the vertex with the maximal out-degree is iteratively selected into the set that stores the vertices representing non-inferior solutions, and all the vertices that have edges from the selected vertex are deleted from the graph.

The comparison results of FSBI, [3], [6] and the deterministic buffering in nominal scenario are shown in Table 3. The solution with the minimal mean value of D (if tied, the solution with the minimal variance) is selected as the final solution. The timing constraints are randomly selected such that our algorithm and the other algorithms have the yields in the reasonable range [60%, 100%]. The approximation of the multiplication of Gaussian variables as a Gaussian variable may have 10% errors on the PDF [6], so we use Monte Carlo simulation to compute the yield for the selected solution. Column 2 shows the timing constraints, Column 3 and 4 show the number of buffers and the yield, respectively, computed by the van Ginneken's algorithm in nominal scenario, Column 5 and 6 show the running time and the yield, respectively, computed by [3], and Column 7 shows the yield computed by [6]. Column 8, 9 and 10 show the number of buffers, the running time and the yield from FSBI, respectively. Column 11 shows the yield improvement of FSBI compared with the deterministic buffering. The "N/A" in the table means that the algorithm cannot finish the test case because of the memory constraint (2GB) or time limit (3 hours).

On average, FSBI achieves 12.34% improvement on the yield compared with the deterministic buffering. The yields from FSBI are always higher than those from the deterministic buffering for all these cases. For example, the FSBI achieves 16.78% yield gain for test case "r1". We also implemented an approach that prunes the solutions according to only the prune rule, so this approach keeps all the "non-inferior" solutions, and thus gives us a good estimation of the optimal yield. It can finish only the case "r1" because of the limitation of the memory. The computed yield of "r1" is 79.47%, which is quite close to the yield from FSBI.

Table 3: Comparison results of FSBI, [3], [6] and deterministic buffering.

Nets		Van-Ginneken (Nominal)		[3]		[6]	FSBI			Gain
Name	D constr	# Buffer	Yield (%)	Time (s)	Yield (%)	Yield (%)	# Buffer	Time (s)	Yield (%)	(%)
p1	805	162	62.88	N/A	N/A	63.88	158	10.22	75.28	12.40
p2	2030	268	79.21	N/A	N/A	73.60	268	27.92	94.37	15.16
r1	335	166	62.60	198.75	77.81	59.10	169	5.26	79.38	16.78
r2	454	358	67.03	N/A	N/A	62.90	363	17.53	79.49	12.46
r3	620	517	81.30	N/A	N/A	79.30	523	14.20	92.48	11.18
r4	900	1187	78.56	N/A	N/A	79.26	1192	52.67	87.26	8.70
r5	1080	1893	70.65	N/A	N/A	71.03	1918	76.63	80.36	9.71
ave										12.34

So the yield is not sacrificed much in FSBI. The results demonstrate that the statistical buffering is still needed for multi-pin nets, and it can use the information provided by the deterministic buffering to achieve higher yields. The yields computed by [6] are higher than those computed by the deterministic buffering that assumes the worst situation (these results are shown in Table 1), which is consistent with the conclusion in [6]. But in general, they are not much higher, sometimes even lower, than the yields computed by the deterministic buffering that assumes the nominal situation. The FSBI always achieves much higher yields than the approach in [6]. The results in Table 3 also indicate that the FSBI is very efficient. It takes only 76.63 seconds for FSBI to finish the largest case "r5". While [3] cannot finish most of the test cases because of the memory constraint (2GB) or time limit (3 hours). Although the approach in [6] is efficient, we do not report its running time because of the quality of its solutions.

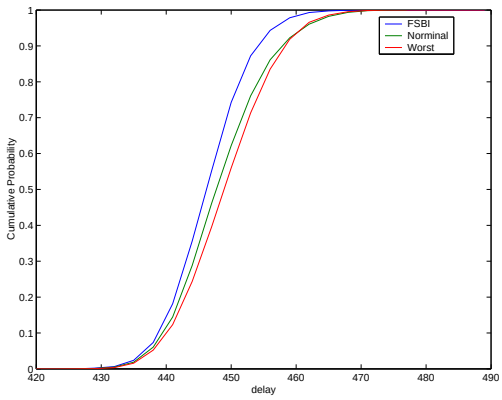


Figure 5: Comparison of solutions for "r2"

Through Monte Carlo simulation, Fig. 5 shows the cumulative distribution functions (CDF) of the maximal delays from the root to sinks for the final solutions from those algorithms. The "Nomina" curve and the "Worst" curve represent the CDFs of the solutions from the deterministic Van-Ginneken's algorithm that assumes all the parameters are at their nominal values or their $\mu + 3\sigma$ values, respectively. The curve representing the distribution from FSBI is obviously pushed to the left side, so FSBI gets a higher yield.

5. CONCLUSIONS

In this paper, we proposed a buffer insertion algorithm with the consideration of process variations. We use the solutions from the deterministic buffering that fixes all the random variables at the nominal values to guide our statistical buffering algorithm. We keep all the non-inferior solutions according to a correlation-aware prune rule when the size of a solution list is comparable with the size of the deterministic solution list, while if the size of the statistical solution list becomes much larger, we pick only a small number of statistical solutions to merge. Our algorithm always achieves higher yield than the deterministic buffering. The experimental results demonstrate that our algorithm can handle large cases very efficiently, and improves the yield more than 12% on average.

6. REFERENCES

[1] H. Chang and S. S. Sapatnekar. Statistical timing analysis considering spatial correlations using a single pert-like traversal. In *ICCAD*, pages 621–625, 2003.

[2] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *DAC*, pages 331–336, 2004.

[3] V. Khandelwal, A. Davoodi, A. Nanavati, and A. Srivastava. A probabilistic approach to buffer insertion. In *ICCAD*, pages 560–567, 2003.

[4] J. Xiong, K. Tam, and L. He. Buffer insertion considering process variation. In *DATE*, 2005.

[5] A. Davoodi and A. Srivastava. Variability-driven buffer insertion considering correlations. In *ICCD*, 2005.

[6] J. Xiong and L. He. Fast buffer insertion considering process variations. In *ISPD*, 2006.

[7] P. Saxena, N. Menezes, P. Cocchini, and Desmond A. Kirkpatrick. The scaling challenge: Can correct-by-construction design help? In *ISPD*, pages 51–58, 2003.

[8] L. Deng and M. D. Wong. Buffer insertion under process variations for delay minimization. In *ICCAD*, pages 317–321, 2005.

[9] L. P. P. P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *ISCAS*, pages 865–868, 1990.

[10] W. Shi and Z. Li. An $O(n \log n)$ time algorithm for optimal buffer insertion. In *DAC*, pages 580–585, 2003.

[11] W. J. Krzanowski. *Principles of Multivariate Analysis*. Oxford University Press, 2000.

[12] D. B. Owen. Tables for computing bivariate normal probabilities. *The Annals of Mathematical Statistics*, 27:1075–1090, 1956.

[13] Z. Li and W. Shi. Fbi: Fast buffer insertion for interconnect optimization. `http://dropzone.tamu.edu/~zhuoli/GSRC/fast_buffer_insertion.html`.