New Algorithms for Gate Sizing: A Comparative Study

Olivier Coudert[†] Ramsey Haddad[†] Srilatha Manne[†]

Synopsys Inc., 700 East Middlefield Rd.
Mountain View, CA 94043, USA

University of Colorado, Dept. of ECE
Boulder, CO 80309, USA

Abstract

Gate sizing consists of choosing for each node of a mapped network a gate implementation in the library so that some cost function is optimized under some constraints. It has a significant impact on the delay, power dissipation, and area of the final circuit. This paper compares five gate sizing algorithms targeting discrete, non-linear, non-unimodal, constrained optimization. The goal is to overcome the non-linearity and non-unimodality of the delay and the power to achieve good quality results within a reasonable CPU time, e.g., handling a 10000 node network in 2 hours. We compare the five algorithms on constraint free delay optimization and delay constrained power optimization, and show that one method is superior to the others.

1 Introduction

Early work on gate sizing targeting area/delay optimization can be found in [20, 12]. Using a RC delay model, TILOS [8] expresses the delay and area as posynomials. Geometric programming or heuristics based greedy approaches can be used to solve such a posynomial formulation [23, 22]. Linear programming is used in [2] thanks to a piecewise linear delay model. A convex programming formulation based on pseudo-posynomial is presented in [22], and is solved using an interior point method. Gate sizing is formulated as non-linear programming in [4, 11, 15] and solved with Lagrangian multipliers [19, pp. 60–74]. Analytical delay/power/area models or continuous sizing are used in [20, 10, 22, 3] to avoid facing the combinatorial explosion, or to fill the lack of first and second derivatives.

These approaches suffer from some of the following problems. (1) The delay and power cost models are no longer realistic, or are over-simplified to fit an optimization technique. (2) Some methods use continuous sizing with the idea of solving an easier problem and then projecting the continuous solution on a discrete solution. But gate sizing is essentially a discrete problem, and projective methods can even fail to find a feasible solution. (3) Some methods make crude assumptions on the optimality criterion, e.g., minimizing a weighted power and delay product is the best delay/power tradeoff, while the problem is about *constrained* optimization. (4) Some methods assume that the objective function and/or the feasible region are convex, which does not hold with accurate delay and power models. (5) Also some of the approaches mentioned above are too CPU intensive to be applied to circuits with more than 1000 nodes.

From the practical point of view, gate sizing consists of optimizing the power and/or area under some delay constraints, or optimizing the delay under some power and/or area constraints. The constraints can also include design rules checking (DRC), such as maximum fanout load or maximum transition time. Accurate delay models make gate sizing a non-linear, non-convex, constrained, discrete, optimization problem. Experiences show that it is not unimodal, i.e., several local extrema exist.

This paper addresses gate sizing as defined above. Section 2 discusses constraint free delay optimization. It describes five different algorithms: genetic, polytope, greedy, Hooke & Jeeves, and GS [5] (Global Sizing). Section 3 addresses power minimization under delay constraints. Experimental results of Section 4 will show that GS is the best of these approaches for delay optimization, and that it is better than the widely used greedy algorithm for delay-constrained power optimization.

2 Delay Optimization

This section addresses constraint free delay optimization. Greedy algorithm is the most widely used method for gate sizing. It iteratively resizes nodes on or near the critical path (see below) of the network using various heuristics [9, 23, 8, 3, 12]. The goal of this paper is to explore new alternatives and show that a better optimization method evolves from these considerations.

We have chosen four highly non-linear optimization oriented methods[1]. We distinguish two optimization strategies, depending on how they find a direction in the search

[†]This work was partially funded by ARPA under contract no. F33615-95-C1627.

[1] We excluded simulated annealing because it does not fit well

DAC 96 - 06/96 Las Vegas, NV, USA

space along which the objective function is susceptible to being improved. The *first strategy* (e.g., genetic and polytope algorithms) consists of using a set of configurations in the search space to determine a search direction. The *second strategy* (e.g., Hooke & Jeeves, greedy, and GS algorithms), exploits local information to determine a search direction from one configuration. Before going through these approaches, we briefly overview delay evaluation.

2.1 Delay Evaluation and Notations

To each point n of a network is associated an *arrival time* $AT(n)$, which is the time at which the signal is propagated from the primary inputs to n, and a *required time* $RT(n)$, which is the time at which the signal must arrive to meet point-to-point delay constraints. The *slack* S is defined as $S(n) = RT(n) - AT(n)$. The set of points that has the minimal slack value constitute the *critical path* of the circuit, i.e., the slowest topological path. If the smallest slack is non negative, the delay constraints are met. The reader is referred to [7, pp. 225–289] for more details on delay computation, path sensitization, and false paths.

The time needed for a signal to propagate from an input of a gate to an input of the next gate depends on the intrinsic delay of the gate, the output load (the output capacitance seen at the output of the gate) and on the input transition time (the time needed by the input signal to achieve its transition). The reader is referred to [1, 24, 18, 13, 21, 14] for the presentation of some delay models. An extensive study of different input transition time sensitive delay models shows that a table lookup approach is more accurate than most of the multi-coefficient (linear, polynomial, or posynomial) approximations [14]. We will use such a table lookup based approach, which is within 3% of SPICE.

For the sake of simplicity, we will consider delay optimization as maximizing the smallest slack of the design. We denote N a network, and n a node of a network. We denote g or g_k a gate, $g(n)$ the current gate implementing a node n, and $G(n) = \{g_1, g_2, \ldots\}$ the set of possible gates to implement a node n. We call a *move* a single gate resizing. A move from g_i to g_j is called a move of distance $j - i$. We denote $g(N)$ the current implementation of a network N. When considering a subnetwork N' of N, we will denote n' the node of N' corresponding with the node n of N.

2.2 1st Strategy: Genetic and Polytope

A genetic algorithm applies several breeding operators to generate new configurations from a population (a set of configurations in the search space), and keeps the best solutions to yield the next population. We use the operators *Crossover*, *Smooth*, and *Mutate*. Function *Crossover* builds a solution with the prefix and the suffix of two randomly selected solutions. The probability of a solution to be chosen increases exponentially with its *fitness*, i.e., how good it is regarding the current population. Function *Smooth* averages together two randomly selected solutions. Function *Mutate* randomly changes a few parameters (i.e., which gate implements a node) of a randomly selected solution.

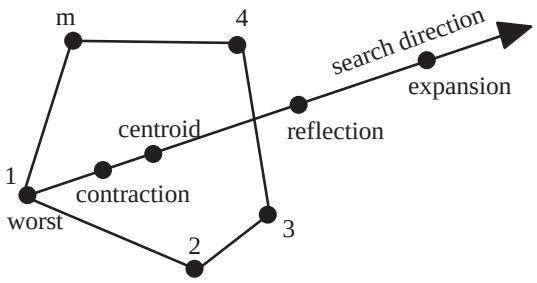

Figure 1: Polytope algorithm.

Fig. 1 illustrates the polytope algorithm [19, pp. 284–292]. It tries to improve the best solution of a set of m configurations (i.e., a polytope made of m vertices in the search space) by iteratively replacing the worst one, which is denoted *worst*, with a better one. To do so, one looks along a search direction that may improve *worst*, i.e., the direction passing through *worst* and the centroid of the polytope. This yields a new solution, *reflection*. If it is better than all the other ones, one looks further in the search direction by computing *expansion*, and the best of these two solutions replaces *worst*. If *reflection* only improves upon *worst*, the latter is replaced with *reflection*. If it does not even improve *worst*, a configuration *contraction* is computed, and it replaces *worst* if it is better. Otherwise, a number of strategies can be used. For example, one determines whether the polytope is tight or loose around the current best vertex, and consequently the other vertices are shrunk towards or bloated away from the current best vertex.

2.3 Issues Regarding 2nd Strategy

The local information that can be used to determine a search direction is the variation of the slack of a node when its gate implementation is changed. One cannot use an analytical delay model because of the lack of accuracy, nor a continuous sizing assumption, which will provide a first order partial derivative, because sizing is essentially a discrete problem. Thus the sensitivity or fitness of a move must be evaluated by delay recomputation.

Sizing a gate effects its load and its output transition times, which effects the propagation time and output transition times of its fanin (because their output loads change) and of its fanout gates (because their input transition times change). Consequently, the propagation time and output transition times of its fanin gates and of its *transitive* fanout gates need to be recomputed. Moreover, sizing changes the sensitivity of the paths, and therefore effects the slack of all the gates. This means that updating the delay after a single move has a cost linear w.r.t. the number of nodes of the circuit, which is computationally too expensive in an iterative algorithm.

Let us denote $\Delta Cost$ (called "gradient") the variation of some function *Cost* when resizing a node. As said

with combinatorial optimization, and is difficult to tune.

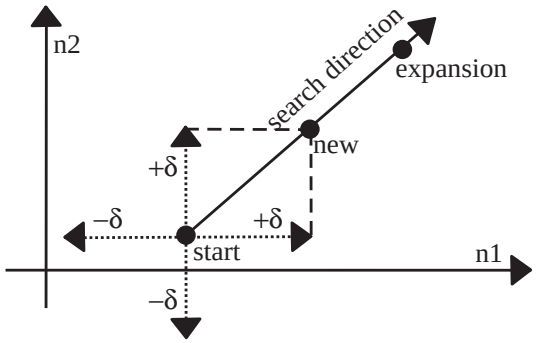

Figure 2: Hooke & Jeeves algorithm.

above, evaluating ΔS of a move must be done within the complete circuit. In practice, although a move can effect the *slack* of every node, its effect on the *slack gradients* decreases quickly, approximately geometrically by fanin and fanout level. This enables us to do two things.

(a) instead of evaluating the gradient of a node n within the whole circuit, which is too computationally expensive even with an incremental delay computation algorithm, we evaluate it within a subnetwork N' extracted around n, made of one or two transitive levels of fanin and fanout. Gradient evaluation becomes cheaper, since it only depends on the richness of the library and on the maximum fanout of the network, regardless of its number of nodes.

(b) after some moves have been performed, the gradient of a node n is re-computed only if n has been sufficiently perturbed, i.e., if one of its close neighbors has been resized. Thus gradient recomputations are avoided for a small loss of accuracy.

Since a local sizing potentially effects the whole circuit, an approach that considers several moves at the same time and that takes into account the interactions of these moves is more "aware" of the global effect of local moves than a single move based iterative method. This raises the need for a global optimization process as opposed for example to a local greedy approach. Moreover, delay optimization can encounter several local extrema because of the non-convexity of the delay model. This raises the need for a method that can avoid being trapped in a suboptimal solution.

2.4 Hooke & Jeeves Algorithm

The idea of the Hooke & Jeeves algorithm [19, pp. 263–276] is to determine a search direction in a multi-dimension space using a minimum amount of information. The local information is the variation of the objective function when moving a parameter by $\pm\delta$. At the beginning, δ is large to allow a wide search, and is then decreased to focus on a local minimum. The composition of the elementary moves that improve the objective function determines the search direction. Fig. 2 illustrates this process in a 2-dimension space.

Let *best* be the current best solution. For each node n of the network *start*, we compute the slack variation (within a subnetwork surrounding n) that results from moving n's current implementation k to the $k+\delta$-th and $k-\delta$-th gate. We record the best of these two moves if one of them yields an improvement. Then all the recorded moves are applied simultaneously to generate the new configuration *new*. If it is better than *best*, it becomes the new *best* solution, and we take as the next starting point the configuration *expansion*, which goes further along the direction that has just improved the objective function. Otherwise, we restart the search from *best*, and δ is decremented to restrict the search to two closer neighbors of each node's current gate implementation.

```
function GoToLocalMin(N, Cost, LocalCost)
update = N;
moves = Ø
loop {
    old_cost = Cost(N);
    foreach n ∈ update {
        Extract subnetwork N' around n;
        init_cost = LocalCost(N');
        n.move = 0;
        n.grad = 0;
        foreach g ∈ Gates(n) and g ≠ g(n) {
            g(n') = g;
            grad = LocalCost(N') − init_cost;
            if (grad < n.grad) {
                n.move = g;
                n.grad = grad;
            }
        }
        if (n.move ≠ 0) {
            moves = moves ∪ {n};
        } else {
            moves = moves − {n};
        }
    }
    moved = ApplyMultiMove(N, Cost, moves);
    update = PerturbedNodes(moved);
} until Converge(old_cost, Cost(N), moved);
```

Figure 3: GS algorithm.

2.5 Global Sizing

Global Sizing [5] (GS) is a mix of a multi-dimension descent based optimization, a perturbation propagation based heuristic that avoids gradient recomputations, and a global perturbation technique to get out of local minimums. We first present the general purpose optimization algorithm, then the delay optimization algorithm.

2.5.1 General Purpose Optimization Procedure

Fig. 3 shows the GS algorithm. It takes as input a network N, a global cost function *Cost* to be minimized, and a local cost function *LocalCost* (so far, assume that it is equal to *Cost*). The set *update* contains the nodes whose gradients need to be computed (initially it contains all the nodes), and *moves* is the set of all nodes that can potentially be resized (initially it is

```
function DelayOptimize(N, S);
best_slack = −∞;
GoToLocalMin(N, −S, −S);
while S(N) > best_slack {
        best_slack = S(N);
        best_sol   = g(N);
        GoToLocalMin(N, −TS, −TS);
        GoToLocalMin(N, −S, −S);
}
g(N) = best_sol;
```

Figure 4: Delay optimization with GS.

empty). For every node n of *update*, the best gradient $n.grad$ and its associated move $n.move$ is computed w.r.t *LocalCost*, using a subnetwork for the evaluation, as explained in (a) above. Then the function *ApplyMultiMove* takes the set *moves* of all non-zero gradient nodes and determines a *multiple* move. This can be done in several way: along the descent direction, or by conjugation of directions [19, pp. 293–327]; by looking at the maximal subset of *moves* that minimizes *Cost*. The set *moved* of nodes that have actually been resized is then returned, from which *PerturbedNodes* derives the new set of nodes whose gradients need to be recomputed at the next iteration, as explained in (b) above.

2.5.2 Constraint Free Delay Optimization

Fig. 4 shows the delay optimization procedure, which takes as input a network N and a slack function S that expresses the timing constraints. We note $TS(N)$ the sum of the slacks of all nodes of the network N. An optimization and a perturbation step are iterated until no more improvement is found. The optimization step consists of maximizing the slack. The perturbation step is used to get out of the local minimum and look for another, potentially better, local minimum, and is indeed another optimization step, namely maximizing TS. Its effect is to *globally* speed up *every* nodes, so that the conflicts between the critical paths are *relaxed*, and the next maximization of $S(N)$ can achieve a better result[2].

3 Power Optimization

Experimental results presented in Section 4 show that GS and greedy beat out the other techniques for constraint free delay optimization. Thus we compare only these two methods for delay-constrained power optimization[3] (it applies as well for area optimization).

3.1 Power Optimization with GS

To minimize the power, a greedy algorithm iteratively applies the move that saves as much power as possible,

```
function PowerOptimize(N, S);
DelayOptimize(N, S);
if S(N) < 0 then S = S − S(N);
GoToLocalMin(N, [−NEG(S), P], [−NEG(S), Relax]);
```

Figure 5: Power optimization under delay constraints.

or the best move of the least critical node, until it cannot move anything without violating a delay constraint.

Fig. 5 shows how GS is used to optimize the power under delay constraints. The function NEG is defined as: $NEG(x) = x$ if $x < 0$, else 0. The notation $[c_1, c_2, \ldots]$ as a cost function means that the cost c_1 is minimized in priority, then c_2, etc[4]. First the delay is optimized. If the delay constraints cannot be met, they are automatically restated so that power is optimized for the best found delay. Second, the power (P) is minimized while enforcing the delay constraints ($-NEG(S)$)[5].

The local cost function *Relax* used for evaluating the gradient balances the gain in power with a delay dependent function ϕ that acts as a benefit/penalty function. It takes into account how much power and slack is won or lost, and on how critical the node is. In this formulation, S_0 is the initial slack, i.e., $\Delta S = S - S_0$. The small constant ϵ and the normalization constant α are precomputed according to the characteristics of the initial network.

$$Relax = (\alpha \Delta P - \epsilon) \cdot \phi\left(\frac{\Delta S}{\epsilon + |S_0|}\right), \qquad \text{where}$$

$$\phi(x) = \begin{cases} 1 + x & \text{if } x \geq 0 \\ \frac{1}{1-x} & \text{otherwise} \end{cases}$$

The ideas underlying this optimization method are as follows. (1) Optimizing the delay gives plenty of alternatives for power optimization, i.e., going far away from the infeasible region makes power optimization less likely to be trapped in a local minimum. (2) Staying in the feasible region during the optimization avoids the complicated tuning of barrier/penalty based constraint optimization method or ping-ponging (both of these constrained optimization methods allow the solution to be temporally in the infeasible region). (3) The power optimization is done within the feasible region by relaxing the delay constraints using a penalty/benefit function, as opposed for instance to a greedy method that resizes as many non-critical nodes as possible to their minimal power. Such a greedy method can be trapped in a low quality local minimum, because resizing a few nodes to their local minimal power too "quickly" creates critical paths that can prevent most of the other nodes from being resized and saving more power.

[2] We tried several perturbation functions, e.g., guarded randomization, (un)guarded sum of outputs' slacks maximization, but none of them were as good as maximizing $TS(N)$.

[3] An overview on power estimation can be found in [16, 6].

[4] $[c_1, c_2, \ldots] < [c'_1, c'_2, \ldots]$ if and only if there is some index k such that $c_k < c'_k$ and $c_j = c'_j$ for $j < k$.

[5] Note that at this point, S is necessarily non negative, thus taking $-NEG(S)$ as a priority cost enforces the minimization to be done within the feasible region.

Example				%delay improvement					CPU				
Name	Nodes	Ave	Space	Gen	HJ	Poly	Greed	GS	Gen	HJ	Poly	Greed	GS
C2670.H.cb60	701	3.76	10^{165}	0.92	0.00	1.57	1.94	2.77	105	21	245	27	32
C2670.H.cmos	674	4.95	10^{192}	6.29	7.14	6.83	6.93	7.14	40	51	102	6	16
C5315.H.cmos	1001	4.82	10^{432}	2.35	3.05	3.27	3.51	4.20	61	48	222	31	43
pair.H.cmos	1202	5.31	10^{610}	3.66	5.36	4.98	5.60	5.36	141	108	340	33	74
C3540.L.ibm	850	7.68	10^{635}	0.26	5.87	4.91	5.61	7.36	43	198	359	97	234
C7552.H.cmos	1358	5.12	10^{695}	1.16	3.59	2.03	2.38	3.53	197	183	385	39	92
C6288.H.lca3	1727	4.02	10^{730}	19.31	16.85	22.17	24.60	24.75	3455	241	7426	2422	477
des.M.cb60	2687	3.84	10^{1132}	0.16	0.00	4.44	5.67	7.23	426	71	5608	349	440
F642925.M.cb60	11692	3.43	10^{3530}	24.44	11.74	21.30	25.55	26.59	6656	141	12053	2023	887
F642925.M.cbc7	11479	4.51	10^{4088}	2.98	9.61	10.26	11.66	11.89	393	143	10320	2918	814
F642925.M.cmos	11408	4.53	10^{4472}	24.25	26.43	24.73	26.50	27.17	3613	250	7763	1084	782
F642925.M.lca3	14136	5.30	10^{4708}	30.18	28.12	28.30	33.43	33.31	8759	171	12029	8604	1496
tandem.M.cmos	15061	4.31	10^{6057}	25.93	26.60	22.73	24.93	27.45	5668	219	16201	5182	616
F642925.M.ibm	20225	6.54	10^{6896}	40.88	44.55	43.06	45.03	48.47	6198	1739	11559	14177	8826
F642925.L.ibm	21174	6.56	10^{7375}	53.68	55.19	17.00	54.40	57.43	10744	3302	9092	30813	13579
tandem.M.ibm	16181	7.90	10^{9484}	45.82	30.53	34.83	45.77	48.21	22010	365	19554	5468	2313

For each circuit, the table gives the number of **Nodes**, the **Ave**rage number of sizes for the sizeable nodes, and the size of the search **Space**. The **CPU** time is in seconds on a 60 MHz SuperSparc (85.4 SpecInt).

Table 1: Comparison of the five algorithms on delay optimization.

Algorithm	Ave	Wins	Eff	CPU
GS	9.51	84	99.8	519
greedy	8.72	6	72.3	1142
polytope	7.19	0	52.8	2797
Hooke & Jeeves	6.64	5	46.2	173
genetic	6.83	0	36.4	1382

The table gives the **Ave**rage percent of delay improvement and the overall **Eff**ectiveness. **Wins** is the number of cases where the algorithm beat or tied all the other algorithms. The average **CPU** time is in seconds on a 60 MHz SuperSparc (85.4 SpecInt).

Table 2: Result summary for delay optimization.

4 Experimental Evaluation

We took 92 different mapped circuits as a benchmark suite. The mapped circuits came from 9 different logical circuits mapped to 5 different libraries with three different delay oriented mapping efforts. We ran the five delay optimization algorithms presented in this paper. Table 1 details some of the results, and Table 2 summarizes the effectiveness of the different approaches. The effectiveness measure (**Eff**) works as follows. The available improvement is the difference between the initial delay and the best delay found by all the algorithms. Each algorithm earns a score based on how much of that available improvement it finds (e.g., it wins 100% when it finds the best delay, and 0% if it does not find any improvement). **Eff** is the average of these scores over all test cases. The motivation for this measure is to equally reward algorithms that do well in cases where there is not a large possible improvement from the initial circuit.

The genetic algorithm fared the worst when using the **Eff** measure, but ranked better by the **Ave** measure. This is because it did well on some examples that had large percentages of available delay improvement, but did poorly on examples with small percentages of available delay improvement. In either case it uses lots of CPU time. The polytope was fairly reliable at getting a medium improvement in the solution, and this helped it beat out Hooke & Jeeves on average. Hooke & Jeeves was more bimodal: frequently getting stuck early in a local minimum, but occasionally breaking out and finding the best improvement of any algorithm. Hooke & Jeeves also is significantly faster than Polytope.

Table 2 shows that GS beats out the other approaches, including the widely used greedy algorithm. For small search spaces, it is hard to declare which algorithm is faster. However, for large search spaces GS is consistently faster. The best-fit polynomial expressing the CPU time as a function of sizeable nodes is like $|N|^{1.75}$ for the greedy algorithm, while it is like $|N|^{1.20}$ for GS, with a crossover point around 1300 nodes.

Fig. 6 shows typical optimal power/delay curves computed with the GS and greedy algorithms. Both of the methods started with the same network (an optimal delay configuration). We then set different delay constraints and let the algorithms optimize the power. Overall, the optimal power/delay curve computed by GS is always better than the one computed by the greedy algorithm, with only a 10% increase in CPU time. The worst difference between the curves range typically from 1% to 7.5% (5.2% on average), and over 25% for some examples.

5 Conclusion

This paper has presented an extensive study of five different approaches for gate sizing. It has shown that a

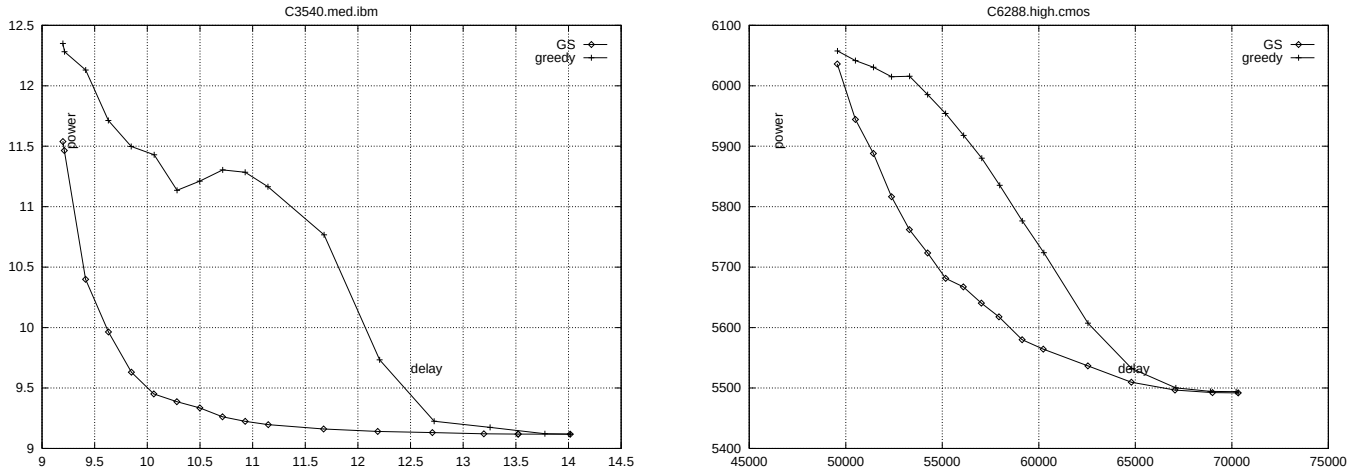

Figure 6: Optimal power/delay curves found by GS and greedy algorithms.

method based on multiple moves, perturbations, and relaxation can achieve better delay and delay-constrained power optimization than the widely used greedy approach. This general purpose optimization algorithm is able to handle fairly large circuits in a reasonable CPU time (10000 nodes in 2 hours) with an improvement in the quality of the result.

References

[1] D. Auvergne, N. Azemard, D. Deschacht, M. Robert, "Input Waveform Slope Effects in CMOS Delays", in *IEEE J. Solid-State Cir.*, **25**-6, Dec. 1990.

[2] M. Berkelaar, J. Jess, "Gate Sizing in MOS Digital Circuits with Linear Programming", *EDAC'90*, 1990.

[3] M. Borah, R. M. Owens, M. J. Irwin, "Transistor Sizing for Minimizing Power Consumption of CMOS Circuits under Delay Constraint", *1995 Int'l Symp. on Low Power Design*, pp. 167–172, Dana Point CA, April 1995.

[4] M. A. Cirit, "Transistor Sizing in CMOS Circuits", *24th DAC*, pp. 121–124, June 1987.

[5] O. Coudert, "Gate Sizing: a General Purpose Optimization Approach", in Proc. of *ED&TC'96*, Paris, France, March 1996.

[6] DesignPower, *Starter Kit*, Synopsys.

[7] S. Devadas, A. Ghosh, K. Keutzer, *Logic Synthesis*, McGraw-Hill, 1994.

[8] J. P. Fishburn, A. E. Dunlop, "TILOS: a Posynomial Programming Approach to Transistor Sizing", *ICCAD'85*, pp. 326–328, Nov. 1985.

[9] J. P. Fishburn, "LATTIS: an Iterative Speedup Heuristics for Mapped Logic", *29th DAC*, June 1992.

[10] B. Hoppe, G. Neuendorf, D. Schmitt-Landsiedel, "Optimization of High-Speed CMOS Logic Circuits with Analytical Models for Signal Delay, Chip Area and Dynamic Power Dissipation", in *IEEE Trans. on CAD*, **9**-3, pp. 236–246, March 1990.

[11] K. S. Hedlund, "AESOP: A Tool for Automated Transistor Sizing", *24th DAC*, pp. 114–120, June 1987.

[12] C. M. Lee, H. Soukup, "An Algorithm for CMOS Timing and Area Optimization", *IEEE J. of Solid-State Cir.*, **19**-5, pp. 781–787, Oct. 1984.

[13] Motorola HDC Series Design Manual.

[14] A. Martinez, "Automated Library Characterization and Timing Model Accuracy Issues when Interfacing to Different CAD Tools", Hewlett-Packard, Santa Clara.

[15] D. P. Marple, "Transistor Size Optimization in the Tailor Layout System", *26th DAC*, pp. 43–48, June 1989.

[16] F. N. Najm, "A Survey of Power Estimation Techniques in VLSI Circuits", in *IEEE Trans. on VLSI Systems*, **2**-4, pp. 446–455, Dec. 1994.

[17] P. Penfield, J. Rubinstein, "Signal Delay in RC Tree Networks", in *2nd Caltech VLSI Conf.*, March 1981.

[18] R. W. Phelps, "Advanced Library Characterization for High Performance ASIC", Texas Instruments, Dallas.

[19] S. S. Rao, *Optimization: Theory and Applications*, Wiley Eastern Ld., 1978.

[20] A. E. Ruehli, P. K. Wolff, G. Goertzel, "Analytical Power/Timing Optimization Technique for Digital System", in *14th DAC*, pp. 142–146, June 1977.

[21] T. Sakurai, A. R. Newton, "Delay Analysis of Series-Connected MOSFET Circuits", in *IEEE J. of Solid-State Cir.*, **26**-2, pp. 122–131, Feb. 1991.

[22] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, S. M. Kang, "An Exact Solution to the Transistor Sizing Problem for CMOS Circuits Using Convex Optimization", in *IEEE Trans. on CAD*, **12**-11, pp. 1621–1634, Nov. 1993.

[23] J. M. Shyu, A. Sangiovanni-Vincentelli, J. P. Fishburn, A. E. Dunlop, "Optimization-Based Transistor Sizing", in *IEEE J. of Solid State Cir.*, **23**-2, April 1988.

[24] G. Zewi, U. Barkai, Z. Becker, J. Ben-Simon, E. Kadar, "An Accurate Slope-Dependent Delay Model", in *TAU'90*, Haifa, Israel, 1990.