Gate Sizing For Cell Library-Based Designs

Shiyan Hu
Department of ECE
Texas A&M University
College Station, Texas 77843
hushiy@ece.tamu.edu

Mahesh Ketkar
Strategic CAD Labs
Intel Corporation
Hillsboro, Oregon 97124
mahesh.c.ketkar@intel.com

Jiang Hu
Department of ECE
Texas A&M University
College Station, Texas 77843
jianghu@ece.tamu.edu

ABSTRACT
With increasing time-to-market pressure and shortening semiconductor product cycles, more and more chips are being designed with library-based methodologies. In spite of this shift, the problem of discrete gate sizing has received significantly less attention than its continuous counterpart. On the other hand, cell sizes of many realistic libraries are sparse, for example, geometrically spaced, which makes the nearest rounding approach inapplicable as large timing violations may be introduced. Therefore, it is highly desirable to design an effective algorithm to handle this discrete gate sizing problem.

Such an algorithm is proposed in this paper. The algorithm is a continuous solution guided dynamic programming approach. A set of novel techniques, such as Locality Sensitive Hashing based solution selection and stage pruning, are also proposed to accelerate the algorithm and improve the solution quality. Our experimental results demonstrate that (1) nearest rounding approach often leads to large timing violations and (2) compared to the well-known Coudert’s approach, the new algorithm saves 9% - 31% in area cost while still satisfying the timing constraint.

Categories and Subject Descriptors
J.6 [Computer-aided Engineering]: Computer-aided Design

General Terms
Algorithms, Performance, Design

Keywords
Gate Sizing, Dynamic Programming

1. INTRODUCTION
Increasing design complexities along with time-to-market pressures and shortening product cycles have mandated a shift in VLSI design from custom crafting to cell library-based design methodologies. This shift raises an increasing need of salient gate sizing techniques which are powerful in performing delay-area trade-off optimizations. A handful set of gate sizing techniques exist, however, most of them handle the continuous gate sizing problem which is based on the assumption that gate sizes can be any values within certain range (see, e.g., [1, 2]). When gate implementations are restricted to discrete sizes, as in reality, the problem becomes much more difficult and very few approaches (see, e.g., [3, 4]) are known.

On the other hand, a large number of realistic cell libraries are “sparse”. For example, when the cell sizes are geometrically spaced instead of uniformly spaced, significant sparseness is introduced. Refer to [5] for some realistic sparse libraries. Geometrically spaced gate sizes are desired because uniformly spaced gate sizes would result in a large number of gate sizes and managing this large volume of data is difficult [5]. Furthermore, it is proven in [5] that under certain conditions, the set of optimal gate sizes must satisfy the geometric progression.

In this paper, we propose a novel gate sizing technique which directly handles discrete gate sizes. As many efficient solutions exist for the continuous gate sizing problem, one might think of obtaining a discrete solution through rounding a continuous solution. This is very fast but often results in large timing violations. In contrast, the method proposed by Coudert [3], which is based on the multi-dimensional descent optimization, directly handles the discrete sizes. However, it has some trial-and-error flavor and thus has room for further improvement. A dynamic programming approach can search solutions more systematically and therefore has the potential to generate high quality solutions. However, it may suffer from substantial amount of computation overhead, which imposes a great challenge to our problem.

The key idea of the new algorithm is to integrate the solution quality of dynamic programming with the short runtime of rounding continuous solution. That is, we narrow down the searching space of dynamic programming under the guidance from a best continuous solution. Thus instead of checking every implementation, our algorithm only investigates a number of discrete implementations around the best continuous solution. This enables us to find solutions with quality close to the best continuous case and at the same time obtain huge speedup in computation. We also develop new techniques to maintain/increase the diversity of intermediate solutions. Focusing on a small number of diversified and representative solutions during the candidate
solution propagation can improve the efficiency of solution search. To this end, an advanced scheme called Locality Sensitive Hashing (LSH) technique [6] is explored to maintain solution diversity via selecting well-spaced solutions in high dimensions.

In summary, the main contributions of this paper are (1) a continuous solution guided dynamic programming approach for discrete gate sizing, and (2) a Locality Sensitive Hashing based solution selection technique that allows obtaining high quality solutions by maintaining solution diversity.

Our experimental results demonstrate that (1) nearest rounding approach often leads to large timing violations for sparse cell libraries and (2) compared to the well-known Coudert’s approach in [3], the new algorithm saves 9% - 31% in area cost while still satisfying the timing constraint.

2. PROBLEM FORMULATION

Given a combinational circuit with \( n \) gate nodes, \( n_i \) primary input nodes and \( n_o \) primary output nodes, a gate library \( L \) consisting of \( |L| \) gate types where each gate type, characterized by the functionality and the number of gate inputs, may have various gate sizes, the discrete gate sizing problem asks to compute a sizing solution with the minimal total gate area such that the maximum delay between any primary input node and any primary output node is bounded above by a delay constraint \( \alpha \). The problem can be formally defined as

\[
\text{Minimize } \sum_{i=1}^{n} a_i W_i \\
\text{s.t. } \frac{\text{Delay}}{C_8} \leq \alpha, \\
W_i \in L.
\]

where \( W_i \) denotes the size of gate \( i \) and \( a_i \) denotes its weighting factor. Weighting factors can be set to unity for gate size minimization, and to weighted summation of signal probabilities and activity factors for explicit power optimization.

3. OPTIMIZATION METHODOLOGY

Before investigating the discrete gate sizing problem, it is helpful to go over the closely-related continuous gate sizing problem. In this problem, gate sizes are allowed to be any real value between certain lower and upper bounds, and the resulting problem can be efficiently solved, for example, by using Lagrangian Relaxation technique [2]. Moreover, an optimal solution can be obtained if the underlying delay model is a posynomial function. For example, the solution is proven to be optimal when the Elmore delay model is used [2].

It might be expected that a good discrete solution can be obtained by rounding the gate sizes of continuous solution to the nearest discrete gate sizes. However, this is not the case for the sparse cell library as the choices of gate implementations are very restrictive (see also [3] for this observation). Our study shows that in a sparse cell library, the nearest rounding strategy can make the ISCAS’85 benchmark circuits to have timing violations of several nanoseconds in 0.13um technology (refer to Table 2 in Section 5).

On the other hand, the dynamic programming approach can obtain the optimal solution for the discrete gate sizing problem, however, it is computationally prohibitive as it needs to investigate every gate size at each gate node.

Our idea is to integrate the optimality of the dynamic programming framework with the high efficiency of the rounding approach. To this end, we propose a continuous solution guided dynamic programming algorithm to solve the discrete gate sizing problem. That is, the searching space of the dynamic programming is significantly narrowed down under the guidance from a good continuous solution while solution quality is only slightly degraded in spite of huge speedup. At each gate node, instead of every discrete gate size, only those close to the continuous solution will be investigated. This difference between the dynamic programming approach (without any speedup technique) and the new scheme enables us to find solutions with quality close to the best continuous case and at the same time obtain tremendous speedup in computation.

Any of the well-known continuous optimization techniques can be used to obtain solution of the relaxed, that is, continuous problem. In this paper we use Lagrangian Relaxation based algorithm [2]. For simplicity, we also adopt Elmore delay model in this paper. However, our method is independent of delay model and any delay model is applicable. For example, the convex delay model [7] can be employed to compute a better continuous solution guider and a better overall result. Rest of the paper concentrates on our novel algorithm used to discretize the continuous solution.

4. DISCRETIZATION ALGORITHM

Before explaining the algorithm, we will present our circuit model and elaborate on some key terms necessary to describe the algorithm. In our algorithm, a circuit is represented as a directed acyclic graph where each node corresponds to either a logic gate or a primary input or output of the circuit, and each edge corresponds to a pin-to-pin connection between two gates. A cut line, shown in Figure 1, is defined as a subset of edges in the circuit such that it partitions the circuit graph into two disjoint subgraphs. Any two cut lines do not cross, and the part between two neighboring cut lines is referred to as a stage.

![Figure 1: Cutlines and a stage](image)

In this paper, a complete solution refers to the determination of all gate sizes in the circuit. A partial solution is a solution where not all gate sizes have been determined. We will denote a discrete solution by \( \gamma \) and the underlying continuous solution by \( \gamma^c \). Outline of the algorithm is described in Figure 2.

The algorithm starts by performing a breadth-first traversal on the circuit graph to identify cutlines and stages. Once the cutlines and stages are identified, the algorithm begins with the primary input nodes, proceeds through a
0. Given: Continuous solution \( \gamma^c \)
1. Algorithm: Discretize GateSize
2. Compute cutlines
3. For each stage from input to output \( i \)
4. Select the best solution at the primary output

5. Algorithm: Process stage \( i \)
6. For each gate \( k \) in stage \( i \)
7. For every compatible assignment of fanouts
8. Compute score
9. Enumerate assignments and prune based on score
10. Perform stage pruning
11. Perform LSH - based pruning

Figure 2: Pseudocode for discretization algorithm

breadth-first traversal of the circuit graph and processes the circuit stage by stage. Note that a cutline is formed if \( t \) gates are traversed during the breadth-first traversal, where \( t \) is experimentally determined to achieve balance between solution quality and runtime for each circuit.

At each gate node, only gate sizes close to the continuous solution are investigated. To this end, possible sizes of immediate fanout gates are first sorted according to their “distance” to the continuous gate assignment, and the top few assignments are selected for investigation. Details of this step are provided in the section Section 4.1. A few candidates from this list are chosen for further investigation by employing a criticality-aware randomized solution selection strategy. This selection is outlined in Section 4.2.

Once a gate node is processed it is included in a partial solution and thus the sizes of it and all its immediate fanout gate nodes have been determined. Inside a stage, solutions are propagated from fanin to fanout, where each solution is characterized by cumulative delay and cumulative gate area. Each solution \( \gamma \) is characterized by a \((D,W)\) pair, where \( D(\gamma) \) refers to the maximum delay from any primary input node to any processed node in \( \gamma \) and \( W(\gamma) \) refers to the cumulative gate area for all processed gates in \( \gamma \). After processing a node, \((D,W)\) will be accordingly updated for each new solution. The set of solutions are propagated from one cut line to the next cut line. Along a cut line, inferior solutions are pruned by employing stage pruning and by an LSH based solution diversification and selection scheme, which are explained in Sections 4.3 and 4.5 respectively.

A partial solution becomes a complete solution when all stages and gates are processed. For convenience, when there is no confusion, we also call a partial solution a solution.

4.1 Score: Proximity to Continuous Solution

Every assignment of library cells to a gate and its fanouts is characterized by a score. For the ease of understanding, we first show how to compute score using Figure 3, and defer the formal definitions for the time being.

Suppose that we proceed to the gate \( g_7 \) which has three immediate fanout gates \( g_7, g_8, g_9 \). Let \( O(g) \) denote the set of immediate fanout gates of \( g \), then \( O(g_7) = \{g_7, g_8, g_9\} \). Suppose that \( g_7 \) has three implementations 1, 5, 10. \( W \) is used to denote the area of each implementation, i.e., \( W_1(g_7) = 1, W_2(g_7) = 5, W_3(g_7) = 10 \) assuming that the gate length is 1. Let \( M_{g_7} \) be the number of implementations for gate \( g_7 \), then \( M_{g_7} = 3 \). Denote the gate size for \( g_7 \) under the continuous solution \( \gamma^c \) by \( \gamma^c(g_7) \). Suppose that in the best continuous solution, \( g_7 \) is implemented with size 6.5, i.e., \( W_7(\gamma^c(g_7)) = 6.5 \). We are now to sort \( g_7 \)’s three implementations, namely, 1, 5, 10, according to our “preference” on them, which is measured by their absolute difference to 6.5.

The resulting list is \{5, 10, 1\}, which is denoted by \( \ell_{g_7} \). A score function \( s_g(\cdot) \) is defined on each implementation and measures our preference of each implementation. Its value is equal to the index of the implementation in \( \ell_g \). Thus, \( s_{g_7}(5) = 1 \) since in \( \ell_{g_7} \), gate implementation of size 5 has the index of 1. Similarly, we have \( s_{g_7}(10) = 2, s_{g_7}(1) = 3 \).

![Figure 3: Illustration of computing score.](image)

We now proceed to Figure 4 and illustrate total score computation. Assume that \( g_8 \) and \( g_9 \) both have two implementations, namely 3, 7 and 6, 10, and \( s_{g_8}(3) = 1, s_{g_8}(7) = 2, s_{g_8}(6) = 1, s_{g_8}(10) = 2 \). Denote by \( \Phi_g \) an assignment of implementations of all immediate fanout gates of \( g \). Suppose that in an assignment \( \Phi_{g_8,1} \) we choose size 5 for \( g_7 \), size 7 for \( g_8 \) and size 10 for \( g_9 \). Then \( \Phi_{g_8,1}(g_7) = 5, \Phi_{g_8,1}(g_8) = 7, \Phi_{g_8,1}(g_9) = 10 \). The total score for the assignment \( \Phi_{g_8,1} \) is \( \sum s_g(\phi_g) = s_{g_7}(5) + s_{g_8}(7) + s_{g_9}(10) = 1 + 2 + 2 = 5 \).

![Figure 4: Illustration of computing total score.](image)
is a parameter and \( S(\phi_g) \) is defined as
\[
S(\phi_g) = \sum_{g_j \in O(g)} s_{g_j}(\phi_g(g_j)).
\] (2)

In gate sizing, it is certainly computationally prohibitive to try every possible assignment for \( O(g) \). Our idea is to give the preference to assignments with small total scores since they indicate that the assignments are “close” to the continuous solution. As such, low-score assignments are first generated until the total number of candidate gate assignments reaches a threshold. These assignments are then sorted in ascending order according to their total scores and are stored in a new list \( \Phi_g \). Note that in dynamic programming, when two gates share some fanouts, sizes of these fanouts are determined in computing assignments for one gate. Later, when another gate is processed, sizes of shared fanouts remain, i.e., only compatible assignments are investigated.

4.2 Selecting Partial Solutions For Investigation

For high efficiency, not every candidate discrete gate assignment in \( \Phi_g \) is tried, rather, only a small portion of it is investigated when processing the gate \( g \). A simple way is to only investigate best assignments (i.e., the assignments with the smallest total scores). However, by this greedy strategy, temporarily worse partial solutions are always eliminated from consideration and the chance of getting better solutions is decreased. Thus, a randomized selection strategy is used in this paper: the top few percent assignments are always investigated and the other assignments are selected with the probability of
\[
Pr = 1 - e^{-\frac{\text{index}(\phi_g)}{\text{index}(\phi_g')}-1},
\] (3)
where \( \text{index}(\phi_g) \) denotes the index of the assignment \( \phi_g \) in \( \Phi_g \). Precisely, we generate a random number (uniformly distributed in \([0, 1])\) for each candidate assignment, and compare it with the \( Pr \) associated with the candidate assignment. The assignment is selected if the random number is smaller. This selection criterion favors gate assignments close to the continuous solution as a small index in \( \Phi_g \) is a sign of proximity to the continuous solution. Note that our randomized selection strategy, which keeps the potential of jumping out of the local optima during optimizations, is in spirit similar to those used in many popular computing schemes such as simulated annealing.

Our algorithm is tuned to be slack sensitive, meaning that more gate assignments will be investigated for timing critical nodes. Since one does not know the slack of each node before completing the sizing procedure, the slack is estimated using our guider, i.e., the continuous solution. We define the normalized slack of a gate node to be the slack of the node over the worst slack of the circuit. Note that we always have continuous solutions with positive slack as long as the timing constraint is not too tight. When the worst slack is very close to zero, a small positive constant is added to the slack of all nodes for the robustness of computation. Clearly, a normalized slack is always no smaller than one and a node is more timing critical with the smaller normalized slack. In our implementation, Eqn (3) is tuned to be as follows
\[
Pr = \frac{1 - e^{-\frac{\text{index}(\phi_g)}{\text{index}(\phi_g')}-1}}{(\text{normalized slack}(g))}.
\] (4)

4.3 Solution Pruning

During solution propagation, even though the search space is judiciously restricted by the use of the continuous guider, at each gate, the algorithm still needs to check and keep several solutions close to the continuous solution. Propagating all solutions is impractical and unnecessary. Many of them are inferior to others and should be pruned to save computation cost. There are two types of pruning/inferiority in our case.

We now introduce the first one. For two solutions \( \gamma_1, \gamma_2 \) generated at the same input gate \( g_1, g_2 \) is inferior to \( \gamma_1 \) if and only if \( D(\gamma_1) \leq D(\gamma_2) \) and \( W(\gamma_1) \leq W(\gamma_2) \). The inferior solution will be pruned and thus only the solution with either smaller maximum delay or smaller total area survives. This type of pruning is called node pruning.

After carrying out the computation for a while, the size of the solution set \( \Gamma \) becomes very large. To maintain efficiency, \( \Gamma \) has to be shrunk before successive computations. For this purpose, several cut lines are computed in the circuit graph, and solution propagation pauses when a cut line is met. The solutions are evaluated there and those inferior to others are pruned. This forms the second type of pruning, called stage pruning. Refer to Figure 1. The pruning is based on a fitness value considering all gate implementations along the cut line. Along a cut line, there are a few gates denoted by \( g_1, g_2, \ldots \). Each solution \( \gamma \) is assigned a fitness value \( f(\gamma) \) which is defined as
\[
f(\gamma) = \sum_{g_i} |D(\gamma(g_i)) - D(\gamma^*(g_i))| \cdot \sum_{g_i} |W(\gamma(g_i)) - W(\gamma^*(g_i))|.
\] (5)

Fitness value of a solution measures its proximity to the continuous solution, and a solution with small such value is preferred. In another word, \( \gamma_2 \) is inferior to \( \gamma_1 \) if and only if \( f(\gamma_1) \leq f(\gamma_2) \) and \( W(\gamma_1) \leq W(\gamma_2) \) in stage pruning.

4.4 Diversify Solutions

Numerous solutions may aggregate along a cut line. After a stage pruning, solutions are often heavily squeezed: resulting solutions have fairly close \( D \) and \( W \) values even though they are not inferior to each other. This is contrary to our intention as “widely spread” solutions may enable us to search in a large space and eventually lead to better solutions. Focusing on a small number of diversified and representative solutions can improve the efficiency of solution search: we do not want to waste time on checking many similar solutions. Furthermore, without solution diversification, partially worse solutions tend to be removed by e.g., stage pruning. In contrast, with solution diversification, temporarily worse solutions might survive and therefore enhance the chance of obtaining better results.

To diversify solutions, we group similar solutions and then select several representative ones, which are those with small cumulative delays, from each group. It should be noted that stage pruning is performed on the solutions inside each group before solution selection, so representative solutions from each group are not inferior to each other. Grouping solutions can be realized through mapping each solution to a high-dimensional vector followed by clustering geometrically close vectors. The mapping is computed as follows.

Assume that each gate node in the circuit is indexed and each gate implementation in the gate library is also indexed, then each gate index corresponds to a distinct dimension and
the coordinate along that dimension is equal to the index of the assigned gate implementation. In this way, a solution is mapped to a $d$ dimensional vector if $d$ nodes have been processed. We are now to compute clusters among the resulting vectors. For a large circuit, one has to cluster vectors in a very high dimension. Although the high-dimensional clustering problem has been intensively studied in decades, it remains as one of the hardest problems in the database research. In this paper, based on a highly effective and efficient nearest neighbor query technique called Locality Sensitive Hashing (LSH) [6], a new technique for solution clustering and representative solution selection is introduced.

4.5 Solution Selection by LSH

In cluster computation and nearest neighbor query, “curse of dimensionality”, which roughly says that the computational complexity of the above two operations increases exponentially with dimension, remains as a notorious problem in database research for a long time. To effectively attack this problem, a new approach, called Locality Sensitive Hashing (LSH), is introduced in [6]. With provable bound on approximations, LSH can efficiently approximate similarity search by hashing technique. The basic idea is to hash the vectors such that the geometrically close (resp. far apart) vectors are hashed to the same (resp. different) bins with large probability. LSH enables us to answer a nearest neighbor query in $O(dm^{1/(1+\epsilon)})$ time over an $m$-point $d$-dimensional database for any $\epsilon > 0$. In addition, an approximate nearest neighbor query can be answered in sublinear time excluding the preprocessing time [6], where given a point set $P$, the problem asks to return a point $p \in P$ such that the distance of $p$ to the query point $q$ is at most $1 + \epsilon$ times the distance from the nearest point in $P$ to $q$. It is shown in [6] that LSH is much more efficient compared to many other methods. Successful applications of LSH include [8] on bioinformatics.

Suppose that $d$ nodes have been processed in each solution $\gamma$ in the solution set. $\gamma$ is first mapped to a $d$-dimensional point $p$, where each dimension corresponds to a node. Suppose that there are $m$ solutions in the solution set. After mapping, there are $m$ $d$-dimensional points, which form the point set $P$. We then embed these points into the Hamming space $H$ with dimension $d' = Md$, where $M$ is the number of available sizes for any cell type in the library. This embedding allows us to perform random sampling on the embedded bit string. For embedding, taking each point $p \in P$, we transform it into a binary vector $v(p) = \llbracket \gamma_M(x_1), \ldots, \gamma_M(x_d) \rrbracket$ where $\gamma_M(x)$ denotes the unary representation of $x$ (i.e., $x$ ones followed by $M - x$ zeroes). For example, in a solution $\gamma$, three nodes have been processed and are assigned sizes of 1, 2, 5 which are the indices of the sizes for each processed gates. The solution is then mapped to a point $p = (1, 2, 5)$. Further suppose that $M = 5$, then $|v(p)|$ has length of $d' = 15$ and $v(p) = 10000 : 11000 : 11111$ since e.g., $2 = 11000$ in unary representation (i.e., 2 ones followed by 3 zeros).

For convenience, $v(p)$ for a point $p$ is treated as a bit string. To measure the similarity between two bit strings, the number of bit-wise difference is a natural choice. However, this is inefficient when dealing with large point set. Thus, we will first perform a dimension reduction mapping to these bit strings through random sampling and then the similarity is measure by the bit-wise difference there. For dimension reduction mapping, we randomly choose $k$ elements from $\{1, 2, \ldots, d'\}$, where each element has equal probability to be chosen, and form a index subset $I = \{i_1, i_2, \ldots, i_k\}$. We then map each point $p$ into $h(p) = <v(p)[i_1], v(p)[i_2], \ldots, v(p)[i_k]>$. For example, if we choose $k = 3$ elements from $\{1, 2, \ldots, 15\}$ as $1, 10, 3$ in this order, then $I = \{1, 10, 3\}$ and $h(p) = 10000 : 11000 : 11111$ since $v(p)[1] = 1, v(p)[10] = 0, v(p)[3] = 0$. Function $h(\cdot)$ is called the locality-sensitive hash function [6, 8]. It is shown in [6] that the probability that two embedded points to have the same hash value (i.e., hashed into the same bucket) is “proportional” to their similarity. Based on this intuitive fact, we are able to build hash tables to support efficient similarity search among a set of points. In the method, to achieve provable approximation bound, instead of a single index subset, we will choose $l$ index subsets and build a hash table for each index subset. Refer to [6] for the further details.

Let $\omega$ denote the maximum bucket size. For each $j \in \{1, \ldots, l\}$, $I_j$ consists of $k$ elements randomly sampled from $\{1, \ldots, d'\}$ according to the uniform distribution. Suppose that we have the specifications $p_1, p_2$ as follows: $p_1$ refers to the probability that a point $p$ geometrically close to $q$ is hashed into the same bucket as $q$, and $p_2$ refers to the probability that a point $p'$ far away from $q$ falls into the same bucket. It is proven in [6] that the specifications can be achieved when setting $k = \log_{p_1/p_2}(m/\omega)$ and $l = (m/\omega)^p$, where $p = \frac{\ln p_1}{\ln p_2}$. Recall that we need to find representative solutions. Given an LSH, each bucket corresponds to a cluster and the representative solutions are those with small cumulative delays in each bucket.

5. EXPERIMENTAL RESULTS

The new discrete gate sizing algorithm, denoted by NEW, is implemented in C++ and tested on an X86 computer with a 3.2GHz CPU and 1G memory. Our test cases are ISCAS’85 benchmark circuits with a 0.13$\mu$m gate library where each type of gate has 10 geometrically spaced sizes.

To judge the efficacy of our discretization algorithm we compare its results with the solutions obtained by simply rounding each size in the continuous solution to the nearest discrete size. In addition, Coudert’s approach [3], which is a well-known discrete gate sizing technique, is also implemented for comparison. In this work, we use total active device width as our area cost function. Comparison results are summarized in Table 1. We make the following observations:

- Nearest rounding always introduces large timing violations for our geometrically spaced cell library.
- Compared to [3], 9% - 31% area cost reductions are obtained by NEW while still improving slack in many cases.
- Runtime of NEW (including computing the continuous solution) is generally within 2× of [3]. This is already very good considering a dynamic programming-style approach is performed. The efficiency comes from our continuous solution guided scheme, LSH based solution selection technique and pruning techniques.

As a byproduct, the proposed algorithm enables us to compute a local delay-area tradeoff curve around the continuous solution. This is achieved through adjusting the program (e.g., Eqn. (4)) to encourage more gate sizes to be
investigated at each gate. Refer to Figure 5 for two resulting curves, which are computed within 5 × CPU time of their implementations for Table 1. For each plot in Figure 5, delay-area tradeoff curves with two different continuous guiders are shown and some results by Coudert’s approach [3] are also shown for reference. The obtained local tradeoff curve can help users get better timing constraint for the circuit. With new timing constraint, users can use NEW to generate better discrete solutions.

As our approach is proposed for cell library based designs, we also inspect how its effectiveness scales with the discontinuities in the library. For this, we select five sizes out of ten geometrically spaced sizes for each cell type to form a sparser cell library. Gate sizing using nearest rounding and NEW are performed with this new cell library and the results are summarized in Table 2. Clearly, significantly more timing violations are introduced by nearest rounding approach compared to the original cell library case, while NEW is able to obtain the sizing solutions satisfying the timing constraints.

Table 1: Comparisons using a library with 10 sizes per gate type. Timing constraints and Slack are in ps. Area refers to Area Cost.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>1000</td>
<td>2</td>
<td>0.16</td>
<td>-1598</td>
<td>15</td>
<td>0.26</td>
</tr>
<tr>
<td>C499</td>
<td>1500</td>
<td>1</td>
<td>0.087</td>
<td>-172</td>
<td>18</td>
<td>0.14</td>
</tr>
<tr>
<td>C880</td>
<td>1100</td>
<td>1</td>
<td>0.17</td>
<td>-1399</td>
<td>2</td>
<td>0.26</td>
</tr>
<tr>
<td>C1355</td>
<td>2200</td>
<td>1</td>
<td>0.45</td>
<td>-2236</td>
<td>8</td>
<td>0.69</td>
</tr>
<tr>
<td>C1908</td>
<td>1900</td>
<td>0</td>
<td>0.15</td>
<td>-358</td>
<td>2</td>
<td>0.24</td>
</tr>
<tr>
<td>C2670</td>
<td>1800</td>
<td>2</td>
<td>0.20</td>
<td>-225</td>
<td>3</td>
<td>0.30</td>
</tr>
<tr>
<td>C3540</td>
<td>3000</td>
<td>0</td>
<td>0.28</td>
<td>-307</td>
<td>9</td>
<td>0.35</td>
</tr>
<tr>
<td>C5315</td>
<td>2250</td>
<td>2</td>
<td>1.49</td>
<td>-2115</td>
<td>21</td>
<td>1.99</td>
</tr>
<tr>
<td>C6288</td>
<td>5700</td>
<td>2</td>
<td>1.62</td>
<td>-875</td>
<td>5</td>
<td>1.35</td>
</tr>
<tr>
<td>C7552</td>
<td>2100</td>
<td>0</td>
<td>0.65</td>
<td>-389</td>
<td>10</td>
<td>0.90</td>
</tr>
</tbody>
</table>

Figure 5: Delay-cost tradeoff curves for optimizing two ISCAS benchmark circuits. For each case, two different continuous solutions (denoted as Guider-1 and Guider-2) are used to guide NEW.

Table 2: Comparisons using a library with 5 sizes per gate type. Timing constraints/slack are in ps. Area refers to Area Cost.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>1000</td>
<td>2</td>
<td>0.16</td>
<td>-1598</td>
<td>15</td>
<td>0.26</td>
</tr>
<tr>
<td>C499</td>
<td>1500</td>
<td>1</td>
<td>0.087</td>
<td>-172</td>
<td>18</td>
<td>0.14</td>
</tr>
<tr>
<td>C880</td>
<td>1100</td>
<td>1</td>
<td>0.17</td>
<td>-1399</td>
<td>2</td>
<td>0.26</td>
</tr>
<tr>
<td>C1355</td>
<td>2200</td>
<td>1</td>
<td>0.45</td>
<td>-2236</td>
<td>8</td>
<td>0.69</td>
</tr>
<tr>
<td>C1908</td>
<td>1900</td>
<td>0</td>
<td>0.15</td>
<td>-358</td>
<td>2</td>
<td>0.24</td>
</tr>
<tr>
<td>C2670</td>
<td>1800</td>
<td>2</td>
<td>0.20</td>
<td>-225</td>
<td>3</td>
<td>0.30</td>
</tr>
<tr>
<td>C3540</td>
<td>3000</td>
<td>0</td>
<td>0.28</td>
<td>-307</td>
<td>9</td>
<td>0.35</td>
</tr>
<tr>
<td>C5315</td>
<td>2250</td>
<td>2</td>
<td>1.49</td>
<td>-2115</td>
<td>21</td>
<td>1.99</td>
</tr>
<tr>
<td>C6288</td>
<td>5700</td>
<td>2</td>
<td>1.62</td>
<td>-875</td>
<td>5</td>
<td>1.35</td>
</tr>
<tr>
<td>C7552</td>
<td>2100</td>
<td>0</td>
<td>0.65</td>
<td>-389</td>
<td>10</td>
<td>0.90</td>
</tr>
</tbody>
</table>

6. CONCLUSION

This paper proposes a new gate sizing approach which directly handles the discrete gate library for realistic sparse cell libraries. The new algorithm is based on the idea of continuous solution guided dynamic programming and uses the locality sensitive hashing technique for speedup. Our experimental results demonstrate that 9% - 31% area cost reduction can be obtained compared to the well-known Coudert’s approach. Furthermore, by our approach, a set of tradeoffs instead of a single solution are obtained which can help users get better timing constraint for the circuit and provide significant freedom to meet design specifications.

7. REFERENCES