PaRS: Parallel and Near-Optimal Grid-Based Cell Sizing for Library-Based Design

Tai-Hsuan Wu, *Student Member, IEEE*, and Azadeh Davoodi, *Member, IEEE*

***Abstract*—We propose Parallel and Randomized cell Sizing (PaRS), a parallel and randomized algorithm and tool to solve the discrete gate sizing (cell sizing) problem on a grid. PaRS is formulated based on an optimization framework known as nested partitions which we adopt for the first time in the computer-aided design area. PaRS uses parallelism from a novel perspective to better identify the optimization direction. It achieves near-optimal solutions (under 1%) for minimizing the total power subject to meeting a delay constraint. The embarrassingly parallel nature of PaRS makes it highly scalable. We show small algorithm runtimes, in at most minutes for large benchmarks featuring over 47 000 cells. We make comparison with the optimal solution which we are able to generate using customized and parallel branch-and-bound implementation on a grid. Consequently, we are able to generate the optimal solution within hours. While the optimal algorithm uses up to 200 central processing units (CPUs) on our grid, PaRS achieves significant speedups and near-optimal solutions using only 20 CPUs. We also study the impact of varying number of CPUs in PaRS. Finally, we discuss a grid-based implementation using the "master–worker" framework.**

***Index Terms*—Cell sizing, cloud computing, combinatorial optimization, nested partitions, parallel optimization.**

I. Introduction

CELL SIZING for library-based designs (discrete sizes) is a popular technique which allows exploring tradeoffs between delay and area or power. Traditionally, cell sizing is first treated as a continuous optimization problem, assuming that the sizes can arbitrarily be chosen from a range [2], [3]. The continuous size solutions are typically rounded to their closest discrete options in the library. This results in deviation in the estimated delay and area/power of the continuous solution. This deviation might be low if the cell sizes are uniformly spaced. However, to effectively explore the area/power-versus-delay tradeoffs, cell libraries are designed to be geometrically spaced [4], which makes the rounding errors significant [5].

Previous works on cell sizing include [6] which is based on multidimensional descent optimization. Recently, Hu *et al.* [5] have proposed a dynamic programming approach to cell sizing. It can systematically search the solution space. The solution quality in [5] is shown to be better than that in [6]. However, the reported runtimes in [5] are very high—even though the search is limited to those cell sizes that are close to the solution generated by continuous optimization. Furthermore, it is unclear how close the solution in [5] might be from the optimal one.

In this paper, we propose Parallel and Randomized cell Sizing (PaRS), a parallel and randomized cell sizing algorithm. It minimizes the total power subject to meeting a delay constraint. PaRS is formulated based on a parallel and randomized metaheuristic known as nested partitions which we adopt for the first time in the computer-aided design (CAD) area. This optimization framework has been proposed in [7] and applied to supply chain management and traveling salesman problems. Effective implementation of nested partitions needs utilizing the properties of each problem, as we do for the cell sizing problem in this paper.

PaRS is an iterative algorithm. At each iteration, the size combinations of multiple cells are explored simultaneously. The "most promising" assignment for the selected cells is then found at that iteration with the aid of random sampling on the remaining cell sizes. We propose a scheme for selecting the target cells as well as a sampling scheme to assess the impact of the remaining cell sizes to achieve effective implementation of nested partitions (i.e., fast-convergence high-quality solution).

After choosing a promising assignment at an iteration, the remaining assignments are added to a region known as "complementary region" (CR) which we expect not to contain the optimal solution. We never discard any portion of the solution regions and always view the entire solution space as a union of the promising and the complementary regions. At each iteration, we also evaluate the CR using random sampling. If we find a variable assignment with lower cost from the CR, we backtrack to a previous optimization stage.

We also generate the optimal solution using custom branch-and-bound (BB) implementation to significantly reduce the search time by avoiding the large-sized infeasible regions. We show the parallel implementation of PaRS and custom BB on a grid of heterogeneous central processing units (CPUs) and discuss the "master–worker" (MW) framework which enables a grid-based implementation [8].

Overall, our contributions in this paper can be summarized as follows:

1) PaRS, implementation of nested partition optimization framework for the cell sizing problem. We propose effective partitioning and random sampling of solution space for the cell sizing problem; we can achieve high solution quality with very few number of CPUs in parallel implementation;

Manuscript received September 16, 2008; revised January 14, 2009, April 22, 2009, and July 14, 2009. Current version published October 21, 2009. This work was supported by Wisconsin Alumni Research Foundation. An extended abstract of this paper with few details was published by the 2008 International Conference on Computer-Aided Design [1]. This paper was recommended by Associate Editor I. Markov.

The authors are with the Department of Electrical and Computer Engineering, University of Wisconsin—Madison, Madison, WI 53706 USA (e-mail: twu3@wisc.edu; adavoodi@wisc.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2028682

2) Custom BB implementation of cell sizing so we can make comparison with the optimal solution even for large-sized circuits;
3) Parallel implementation of the aforementioned two is done on a shared grid of CPUs using the MW software framework and Condor system for dynamic and opportunistic utilization of computing resources.

While the number of CPUs in BB dynamically changes during optimization (reaches almost 200 in our simulations), in PaRS, we always fix the number of CPUs *a priori* and show that only 20 CPUs are sufficient to generate near-optimal solutions. The runtime of PaRS is in seconds, whereas the runtime of BB is in hours. Compared to other works, PaRS provides high solution quality and fast runtimes due to an embarrassingly parallel nature.

We would like to emphasize that many of the existing algorithms for CAD applications have serious limitations for parallelism. Consequently, we argue that comparing the runtimes of parallel PaRS with an alternative that can hardly be parallelized is fair. The embarrassingly parallel nature of PaRS is an advantage over many existing algorithms. Our reported runtimes are yet conservative because of Condor [9], the program that manages the use of CPUs on our grid. Condor's philosophy is to use idle cycles of a CPU when the owner is away. If the owner returns, the job is migrated.

Finally, we verify by simulation that the optimal solution for cell sizing is highly correlated with the one for continuous sizing. This confirms the correctness of the assumption made by Hu *et al.* [5] to locally search around the continuous solution. We also bias our search toward the continuous solution in our simulation results for PaRS. However, our algorithm is generic and can be used for searching without any bias.

In this paper, cell sizing refers to discrete sizing problem. We refer to the continuous case as the continuous sizing problem.

In the following sections, we start with an overview of cell sizing in Section II. In Section III, we introduce nested partitions. In Sections IV and V, we discuss the formulation of PaRS and BB for cell sizing. In Section VII, we discuss parallelization for grid-based optimization and an overview of MW software implementation for cell sizing. We end with the simulation results and conclusions.

II. Problem Formulation and Previous Work

Given a combinational circuit with n cells, let w_i denote the size of cell i. Assume $w_i \in Q_i$, where Q_i is the set of discrete sizes for cell i provided by the standard cell library. In the cell sizing problem, we would like to minimize the linear combination of the total area and power of the circuit which can be expressed as a weighted summation of the individual areas and powers of the cells. We require the total circuit delay $D_{\rm cir}$ to not exceed the target delay constraint $D_{\rm tar}$. This optimization problem is formulated as follows:

$$\min_{w_i} \sum_{i=1}^{n} C_i \tag{1}$$
$$\begin{cases} C_i = P^{\rm dyn}(w_i) + P^{\rm stat}(w_i) & \forall\, i \\ D_{\rm cir}(\vec{w}) \le D_{\rm tar} & \\ w_i \in Q_i & \forall\, i. \end{cases}$$

Here, $D_{\rm cir}$ can be expressed in terms of the delays of the individual cells. The cell delays, in turn, depend on the cell size variables $\vec{w} = \{w_1, w_2, \ldots, w_n\}$. Here, C_i is the cost of each cell and is the summation of its dynamic and static powers. Note that $P_i^{\rm dyn}$ implicitly includes the necessary activity factors for dynamic power estimation. Furthermore, $P_i^{\rm stat}$ implicitly represents a weighted summation of leakage for different input vectors where the weights represent the common cases found via simulation (see [10] for more details on power modeling for cell sizing).

Both $D_{\rm cir}$ and C_is are calculated for a given assignment of $\vec{w}$. We also incorporate other parameters that affect the delay and cost such as capacitive loadings and slew rates, as follow. The capacitive loading of each cell in the circuit is computed as the summation of the capacitive loadings of its fan-out cells, which is read from the standard cell library. The delay of each cell is then proportional to its computed capacitive loading. The output slew is also modeled as a function of the input slew for each cell. The standard cell library provides dependence of the output slew on the input slew for each cell type, for example, as a lookup table format. For a given size combination, during static timing analysis, the input slew of each cell in the circuit can be computed, using which its output slew can get computed as well. The final output slew in the primary output gets incorporated in the circuit delay. We further explain our slew consideration in the simulation results section after we discuss our framework for solving the cell sizing problem. Wire delays, if available, are also incorporated in the calculation of $D_{\rm cir}$ and are constants. Finally, the formulation for combinational circuits can be extended for sequential circuits [11].

To solve this optimization problem, many previous works first obtain continuous convex delay and power models for individual cells [12]. This can be done by fitting the data from the discrete library to predefined convex forms and assuming that w_i is a continuous quantity [12], [13]. The obtained formulation will then describe the (continuous) gate sizing problem which can efficiently and optimally be solved using convex programming techniques such as Lagrangian relaxation [3], [11]. The generated continuous solution is discretized to the cell sizes that are available by the library. Reference [5] extends the work in [6] and solves the cell sizing problem. It uses an intuitive observation that the optimal cell sizes should be close to the sizes generated by the continuous sizing problem. In our simulation results, we verify that this is a correct intuition. In this paper, we will solve this discrete optimization problem without any relaxations.

III. Overview of Nested Partitions

Nested partitions [7] is a hybrid optimization framework that combines adaptive global sampling with formal methods and local heuristic search. It uses a flexible partitioning method to divide the search space into regions that can be analyzed individually and then evaluates the results from each region to determine how to proceed with the search. The underlying idea of nested partitions is to focus the computation effort on a subset of the search space that is deemed "promising" while also keeping an eye on the rest of the search space. We denote the promising region by PR.

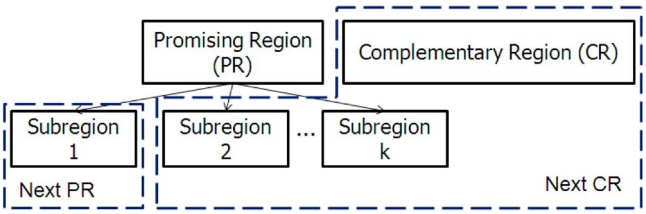

Fig. 1. Overview of optimization using nested partitions.

This idea is realized as follows: The search space is always viewed as the disjoint union of a PR and a CR, as shown in Fig. 1. This is in contrast to the BB algorithm in which the inferior (but possibly feasible) branches are pruned out. At each iteration of nested partitions, we apply the following four steps:

1) **Partitioning:** The current PR is partitioned into k subregions (see Fig. 1);
2) **Random Sampling:** After defining the subregions, we randomly make feasible assignments to the unassigned optimization variables. Each such assignment is referred to as a sample. At each iteration, we sample from the entire solution space: We sample each subregion as well as the CR;
3) **Find Promise Index:** Using the sampling information, each subregion is assessed for its potential to contain a "good" solution. Furthermore, the current CR is assessed for the same criterion. The region that is assessed to have the highest potential to contain a good solution is then designated as the next PR, and the remaining regions are bundled together to form the next CR (see Fig. 1);
4) **Backtracking:** If the partitioning and assessment schemes are effective, the next PR is expected to often be among the subregions of the current one, resulting in the PR to get smaller with each iteration (hence, the word "nested partitions"). However, at a given iteration, it is possible that the CR is assessed to be more promising. In such a case, we backtrack to a previous optimization stage, for example, to the previous iteration, and redo the partitioning with the information hitherto obtained.

Some of the highlighting features of nested partitions are summarized as follows. First, the backtracking feature avoids getting trapped at local minima. Second, the evaluation of the different subregions at each iteration is done in parallel. The sampling step can also be parallelized because the evaluation of each sample is independent from the others. Third, existing heuristics and formal methods can get incorporated in nested partitions—for example, to define a partitioning or sampling strategy. Successful demonstrations of nested partitions exist which combine general-purpose heuristics (like genetic algorithms) and special-purpose ones (heuristics for traveling salesman problem [14]). Indeed, it can be thought of as a metaheuristic that guides the search through a hybrid combination of various types of algorithms [15]. This is an important feature to achieve parallel CAD because it can build upon the existing tools and strategies.

Depending on the target problem (e.g., traveling salesman [14] or product line management [16]), implementation of nested partitions should be obtained by carefully defining how partitioning, random sampling, promise index, and backtracking should be done by utilizing specific features of the problem. This allows effective implementation of nested partitions with fast convergence (few backtracks) and high solution quality. Next, we discuss how this effective implementation can be achieved for the cell sizing problem.

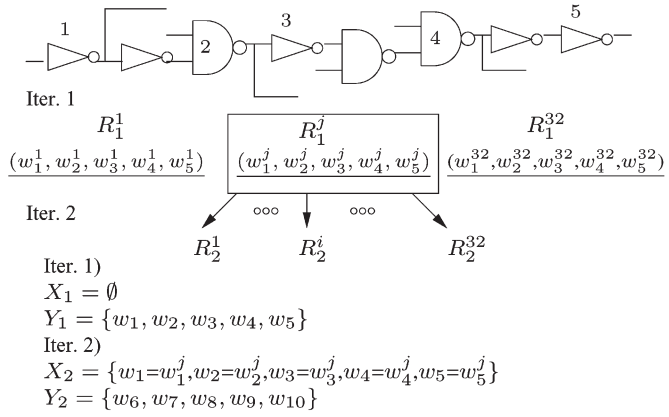

Fig. 2. Exploring the size combinations of $k = 5$ new cells at each iteration for $q = 2$ sizes per cell (hence, 32 subregions). Here, at iteration 1, we select subregion i as the PR.

Please note that nested partitions is a metaheuristic and it has not been proven to converge to the optimal solution in the general case; although under specific assumptions, [17] proves that the framework asymptomatically converges to the optimal one.

IV. Nested Partitions for Cell Sizing

A. General Overview

Fig. 2 gives the general overview of cell sizing using nested partitions. In the first iteration, we start by selecting a subset (k) of the total number of cells (n). We consider all the size combinations of these k cells in the first iteration. If q sizes are available for each cell, we consider all the q^k possibilities, each as a separate subregion. As an example in Fig. 2, we consider the cells indexed 1–5 in the circuit $(k = 5)$ and consider their size variables given by w_1–w_5. If only two sizes are available per cell (so $q = 2$), we will have 32 different size assignments to these five cells, hence 32 subregions.

At iteration i, we denote subregion j with R_i^j. Therefore, in the first iteration, the 32 subregions are denoted by R_1^1–R_1^{32}. Each subregion has one size assignment to the cell size variables. For example, for the jth subregion R_1^j, we will have assignments w_1^j–w_5^j to w_1–w_5, respectively, where the assignment values are one of the two available sizes.

Using the help of random sampling, we may choose one of the subregions to be the next PR at iteration 1. For example, in Fig. 2, subregion R_1^j is selected as the next PR. Therefore, we fix the assignments of variables w_1–w_5 as given by R_1^j and go to the next iteration where we consider the next five cell variables w_6–w_{10}. More formally, we define the following notations.

1) At the beginning of iteration i, let X_i indicate the set of variables that are already "processed". The size variables of all the cells which were considered at the previous

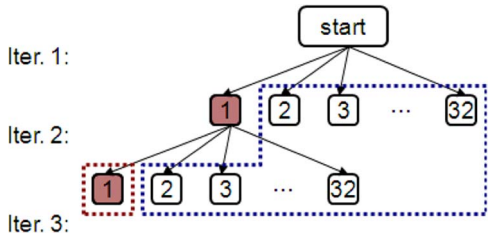

Fig. 3. CR is stored as the union of the subregions which were not selected in all the previous iterations.

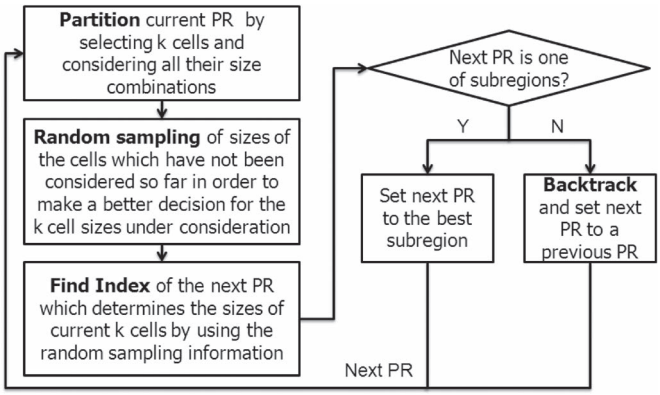

Fig. 4. Overview of cell sizing using nested partitions.

iterations are in X_i (e.g., see Fig. 2 for X_1 and X_2 in iterations 1 and 2).

2) Let Y_i indicate the variables that are "in process" at iteration i. These correspond to the cells that are currently considered for sizing as shown in Fig. 2. Note that $|Y_i| = k$.
3) Let Z_i indicate the "not-processed" variables. These are the remaining cell sizes that have not been considered so far for sizing. Therefore, $|Z_i| = n - |X_i| - |Y_i|$.
4) Let R_i^j indicate the assignment of $X_i \bigcup Y_i$ variables at iteration i given by subregion j. R_i^j has two components:
   a) Assignment of X_i propagated by $R_{(i-1)}^b$, where b is the index of the PR at iteration $i-1$;
   b) Assignment of Y_i given by subregion R_i^j.

At iteration i, once we define the subregions, we consider the remaining assignment to all the cell size variables to be the CR. Fig. 3 illustrates this point for the ten cells selected in the first two iterations. For example, in iteration 1, subregion R_1^1 is selected and divided into 32 subregions in iteration 2. Then, in iteration 2, subregion R_2^1 is selected. Therefore, in iteration 3, the CR is composed of subregions R_1^2–R_1^{32} from iteration 1 which were not selected, along with subregions R_2^2–R_2^{32} from iteration 2. Therefore, the CR not only captures the assignments of the unprocessed variables but also captures the assignments of the processed variables that were not selected. Therefore, the CR is stored as the union of the subregions of previous iterations which were not selected.

Given the definitions of PR and CR, the overview of our cell sizing technique is given in Fig. 4.

At each iteration, we take the current PR and divide it into many subregions by selecting k cells and exploring all their size combinations. We then apply random sampling from the set of unconsidered cell sizes which includes each of the subregions

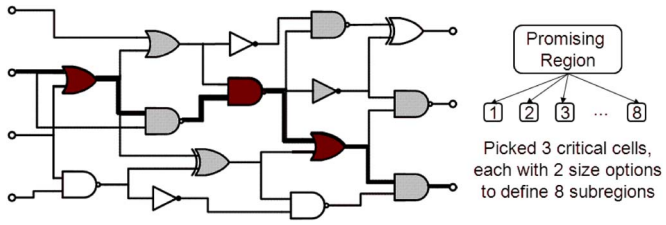

Fig. 5. Cells with highest CF on the "current" critical path are chosen to define the subregions during partitioning. The (lightly colored) neighboring cells will be used for sampling.

and the CR. This helps to assess the most promising size assignment for the k cells under consideration by incorporating the impact of the remaining cells. Using the random sampling information, we find the index of the next PR. If this is one of the existing subregions, we continue by selecting the next k cell sizes in the next iteration. However, if this index is found to be the CR, we need to backtrack to a previous PR.

Effective implementation of the aforementioned framework for cell sizing is not a straightforward procedure for the following reasons: At iteration i, since we explore all the size combinations of Y_i, we hope one of the subregions to be the optimal assignment. *However, since we still do not know the impact of the remaining cell sizes (particularly at the first iterations), it would be challenging to choose the optimal assignment, and, if the next PR is not chosen correctly, many rounds of backtracking might happen, making the runtime of the algorithm very long.*

Therefore, one of our contributions in applying nested partitions for cell sizing is effective partitioning and sampling scheme to decrease the number of backtracking and increase the likelihood of correctly choosing the PR. We elaborate in future sections as we discuss these steps in detail.

B. Steps

Before discussing the steps in nested partitions for cell sizing, we would like to introduce the "criticality factor" (CF) which is a quantity defined for a cell. This CF is used both during our partitioning and sampling steps to order the cells. At iteration i, for cell $j \in Y_i \bigcup Z_i$ (the in-processed and not-processed cells), we define the CF, CF_j as

$$CF_j = \frac{\delta_{D_j}}{\delta_{C_j}} \times \left(1 - S_j^N\right). \tag{2}$$

Here, S_j^N is the slack of cell j—difference between its arrival time and its required arrival time—which is normalized to be between zero and one over all the cells in Y_i and Z_i. To compute the slack, a timing analysis is conducted by fixing some of the cell sizes based on the current region under consideration while setting the remaining ones to their minimum size, as we will elaborate later. Furthermore, in the expression, δ_{D_j} and δ_{C_j} reflect the change in delay and cost of cell j for sizes $w_j^{\max}$ and $w_j^{\min}$, respectively. Our criticality expression is a variation of sensitivity-based factor which has been used in several cell sizing papers such as [10] and [18].

1) Partitioning: As Fig. 5 shows, we first find the most timing critical path (given in bold) from which we select the

cells that are considered in the current iteration. To find the most critical path, we fix the sizes of the cells that were considered in the previous iterations and temporarily set the sizes of the remaining cells to minimum. We then select the k cells for partitioning which have the highest value of CF. During partitioning, we make the size combinations of these k cells to be our subregions. As we will show later, selection of the cells from the critical path allows easier identification of the promising size assignment because we only need to assess the impact of the remaining cell sizes on timing from a small neighborhood.

Algorithm 1 explains the partitioning procedure. We identify the critical path (denoted by Pl_i) in the circuit if each $w_j \notin X_i$ takes its minimized size $w_j^{\min}$ and $X_i = R^b_{(i-1)}$. This gives the minimum cost assignment for $X_i = R^b_{(i-1)}$ which may or may not be feasible. If the delay of Pl_i is larger than the timing constraint (D_{tar}), we need to proceed with sizing and choose Y_i as the k cells with the highest value of CF.

Algorithm 1: Partitioning at iteration i
Input: X_i, $R^b_{(i-1)}$ (processed variables and their assignment)
Output: Y_i: the k selected variables and the "stop" flag
1) Keep $X_i = R^b_{(i-1)}$; stop = true.
2) $\forall\, w_j \notin X_i$, temporarily set $w_j = w_j^{\min}$.
3) Identify Pl_i, the longest timing path at iteration i.
4) If $Delay(Pl_i) > D_{\text{tar}}$:
Set Y_i to the size variables of the k cells with the highest values of criticality ($w_j \in Y_i$ if $j \in Pl_i$ and $w_j \notin X_i$); stop=false.
5) Return Y_i and the stop flag.

The algorithm returns the "stop" flag which is set to true only if the delay of Pl_i is feasible, assuming $X_i = R^b_{(i-1)}$ and minimum size Y_i. In this case, Y_i is the optimal assignment because it yields to a feasible solution with minimum cost. Note that the implicit assumption here is that minimum size results in minimum cost. We will discuss this assumption later. These X_i and Y_i assignments are possibly the global optimum, if they are not found to be suboptimal during the sampling of the CR. We use the "stop" flag to define the stopping criteria.

In the aforementioned algorithm, please note that, since all our cells are selected from the critical path Pl_i, they all have the same slack. Consequently, in the definition of CF given in (2), these cells will have the smallest slack (or a normalized slack of zero). While this might seem a redundant term to have in the definition of criticality, we note that we use the same expression during sampling in which we also consider the cells with different slack values which might not be on the critical path. Therefore, in effect, during partitioning, the cells on the critical path are solely ordered based on the change in delay per change in cost from the current critical path. Please note that, if we consider two cells of the same type on the critical path, they might still have different change in delay because they might have different output loadings which we consider during our computation of CF.

Furthermore, note that, even though the CF is just a linear approximation of the actual dependence of the cell delay with respect to its cost (which, in reality, can be highly nonlinear), it just provides an ordering of the cells on the current critical path, to be considered for partitioning at the current iteration; a less-effective ordering might then translate into more iterations in nested partitions but can still yield to a good solution. Furthermore, we can reduce this error by considering only two consecutive sizes per cell to compute the CF. These two consecutive sizes can be the ones that are immediately before and after the sizes generated using continuous sizing. As we will discuss later, considering these two options during sizing can still generate high-quality results.

2) Random Sampling: This step is applied after partitioning at each iteration. We apply the following two types of sampling: 1) sampling each of the q^k subregions and 2) sampling the CR. The objective of sampling is to get an estimate of the minimum cost solution that is attainable at each subregion, as well as at the CR.

This might require many number of samples, particularly at the first few iterations of nested partitions, since the majority of the cells have not been sized yet. If the sampling information is not a good representative of the attainable solution quality of each subregion, many rounds of iterations and backtracking can happen and significantly increase the execution runtime.

Our contribution here is to reach the practical implementation of random sampling by reducing the number of samples and yet correctly comparing the solution quality of different subregions. To achieve such a sampling, we take advantage of our partitioning scheme in which the cell sizes defining the current subregions belong to the same currently-critical path. We use the intuition that the impact of the cell sizes in current subregions can be assessed using the sizes of the cells which are in a small neighborhood around the current critical path. Therefore, only the cells that are included in this neighborhood should be sampled, and the number of samples can greatly be reduced. Fig. 5 shows this intuition. Next, we will discuss the sampling of the subregions and of the CR.

Sampling the subregions: To sample the subregions, we first identify only a subset of the unprocessed Z_i variables for sampling. We denote this subset by S_i. The remaining Z_i variables will be set to their minimum size during sampling. Algorithm 2 shows the selection of S_i. We first identify a set of primary outputs O, to which the current X_i and Y_i variables connect to. We then select all the nodes in the subtrees of O that are in Z_i.

Algorithm 2: Finding candidate cells S_i for sampling at iteration i
Input: X_i, Y_i, Z_i (processed, in-processed, not-processed variables)
Output: S_i candidates selected for sampling
1) $S_i = \emptyset$.
2) Find a set of primary outputs $\{O | j \overset{P}{\rightsquigarrow} O \,\forall\, w_j \in X_i \bigcup Y_i\}$.
3) Find cells $\{l | l \overset{P}{\rightsquigarrow} O\}$ and update $S_i = S_i \bigcup w_l$.
4) $S_i = S_i \bigcap Z_i$.

The selected S_i for sampling represents the timing critical part of the circuit (critical paths and their neighborhood). This is because X_i and Y_i are chosen to be the most critical cells of the most timing critical path (when the remaining sizes are

set to minimum). The remaining cells in Z_i which are not in S_i are set to a minimum size during sampling to have the smallest contribution to cost without affecting the feasibility. *A particular challenge during sampling is to obtain feasible samples. Since the set S_i is in the timing critical portion of the circuit, it is possible that a random assignment to S_i violates the timing constraint. Our sampling scheme can effectively generate samples with low cost which are all feasible.*

Algorithm 3 shows the sampling of a subregion j at iteration i. It returns an estimate of the minimum cost which is attainable at region j. This minimum cost is denoted by RC_i^j.

Algorithm 3: Sampling subregion j at iteration i
Input: X_i, Y_i, Z_i, S_i, j, and M, the number of samples
Output: RC_i^j (estimate of the minimum cost solution of subregion j)
1) $X_i \bigcup Y_i = R_i^j$ and $RC_i^j = \infty$
2) $w_l = w_l^{\min} \forall\, l \in Z_i$
3) For M steps
  While $D_{\text{cir}}(\vec{w}) > D_{\text{tar}}$
    Select $l \in S_i$ with maximum CF_l
    $r = RAND(0,1)$
    If($\text{r} \leq CF_l$) //select cell size closest to random r
      $f = round(r \times q) \rightarrow w_l = w_l^f$
    $S_i = S_i - w_l$
  Compute $Cost(\vec{w})$ and $RC_i^j = \min(RC_i^j, Cost(\vec{w}))$
4) Return RC_i^j

We first set $X_i \bigcup Y_i = R_i^j$ and Z_i to their minimum sizes. At this point, we have an infeasible assignment. We then apply sampling in M rounds. At each round, we randomly increase the size of the cells with the highest CF [see (2)], one by one until the resulting circuit timing becomes feasible. We record the corresponding cost and return the minimum cost over all the M samples (i.e., the best sample at a subregion). Later, we compare the best samples of different subregions and CR to decide the next PR. This sampling procedure can quickly identify feasible and low-cost samples by starting from the minimum-cost state based on the current variable assignment and randomly upsizing the cells in the order of their CF to reach the feasible solution.

Sampling from the CR: Fig. 3 shows that the CR is stored as the union of the subregions which were not selected in the previous iterations. When sampling the CR, we can sample each of the subregions inside the CR with uniform probability. Considering one subregion inside the CR which might have been encountered at iteration $j < i$, the assignment of X_j and Y_j is known and stored as part of the CR. We can fix those and then sample the remaining cells in that subregion. This procedure samples the entire CR.

We can sample all the subregions inside the CR with uniform probability. However, as we proceed to more iterations, the size of the CR also grows, and, consequently, we would require a very large number of samples to achieve a good assessment of the CR. Therefore, in our implementation of nested partitions, we adopt a more limited sampling scheme of the CR which is explained next.

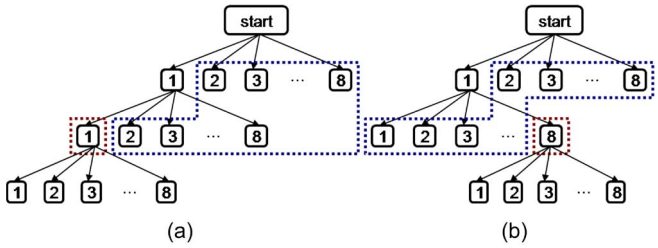

Fig. 6. During backtracking, the current PR (a) is set to the new PR (b). This new PR is inside the CR and is subregion 8 of the previous iteration.

For the CR, at iteration i, we fix the X_i variables and only sample the Y_{i-1} and Z_{i-1} variables. We first sample the Y_{i-1} variables and make sure that the value of Y_{i-1} is different from the assignment of the PR. Once we find a valid assignment for Y_{i-1}, we proceed with the sampling of the Z_{i-1} variables. This gives one sampling (iteration) for the CR.

Consider Fig. 6(a). Here, "start" is iteration 1 in which we define eight subregions. The current iteration is three in which the PR 1 is divided into eight subregions. To sample the CR, we fix the X_2 variables. This corresponds to the assignment of subregion 1 at iteration 2. We first sample the Y_2 variables, making sure it is different than the assignment of PR 1 at iteration 3 (which was subregion 1 at iteration 2). We then generate samples for Z_2 variables. We collect M samples that are similar to the PR. In effect, we sample subregions 2–8 at iteration 2 one more time and compare if their best sample is any better than subregions 1–8 at iteration 3. This procedure allows allocating enough number of samples to the subregions by only sampling "the lowest level" of subregions inside the CR. As we will discuss later, during backtracking, it is still possible that we reach the other portions of the CR in the future iterations.

3) Promise Index: During sampling, we record the minimum-cost feasible sample. We denote this assignment by R_i^{CC}, where CC is the index which refers to the CR. The minimum cost obtained in the CR via sampling is denoted by RC_i^{CC}. The index of the next PR denoted by b is then found from the equation as follows:

$$b = \arg\left(\min_{j=\{1,2,\ldots,k,CC\}} RC_i^j\right). \tag{3}$$

If $b \neq CC$, we set subregion b as the next PR. We combine the remaining subregions and the CR to be the next CR.

4) Backtracking: If $b = CC$, we look up the subregion inside the CR which gave the best sample. We set X_i as given by R_i^{CC} and move to a new partitioning step at the next iteration. Fig. 6 shows the backtracking procedure. In this example, we backtrack from PR 1 at iteration 3 [Fig. 6(a)] to the CR. We backtrack to subregion 8 of iteration 2 and set it to the next PR [Fig. 6(b)]. We then proceed to the next iteration by defining new partitioning as shown in Fig. 6(b).

Recall during sampling that we always sample from those subregions inside the CR which were stored in the latest iteration (lowest level). Using this procedure, we can only backtrack to the subregions which were one level above. If backtracking occurs, during sampling of the next iteration, it is possible

to further backtrack to a subregion from a higher level. In summary, if backtracking becomes necessary, we backtrack to only one level higher; however, if further backtracking becomes necessary in future iterations, it is possible to reach the higher levels of the nested partitions tree. *Our backtracking procedure allows allocating enough number of samples because the number of subregions inside the CR which are considered for sampling is always fixed (eight in our example).* We will further discuss backtracking in the simulation result section.

C. Stopping Condition

If, during an iteration, the "stop" flag is set to true by algorithm 1, we have found the minimum cost assignment for a given X_i. We then apply random sampling and find the promise index. If the promise index is not the CR, PaRS terminates. If the aforementioned never happens, we continue the iterations until all variables are processed. Therefore, in the latter case, iteration i is the last iteration if $|Z_i| = 0$.

D. Current Limitations and Future Extensions

Our specific implementation of nested partitions tailored for cell sizing is due to the following three aspects: 1) partitioning based on the selection of cells from the current critical path; 2) sampling from a small neighborhood around the current critical path; and 3) backtracking to only one level higher inside the CR to reduce the number of required samples.

During partitioning and sampling, we can better assess the impact of the size combinations on delay and cost, because of selecting those cells for sizing which are the more critical ones on the same critical path. More importantly, by focusing on a critical path, the impact of the remaining cell sizes on the circuit delay should only be considered within a small neighborhood around the critical path. This allows significant decrease in the number of required samples.

For cost evaluation for each sample, the remaining unfixed cell sizes outside the neighborhood of the critical path will be set to their minimum sizes. This is based on our implicit assumption in this paper that smaller size corresponds to smaller cost. This generates two limitations in our framework which we will discuss here.

The first limitation is that, by constraining the sampling to a small neighborhood, the overall cost that is attainable at each subregion might get incorrectly assessed. This is because we set the cell sizes outside the neighborhood to the smallest size during sampling based on the assumption that smaller size will result in smaller cost. This helps us to compute the minimum cost that can be introduced by the unsampled cell sizes in the computation of the overall cost. This assumption might not be true depending on the cell library or technology. Furthermore, due to slew constraints for some cells, it might just not be possible to assign the smallest cost to the cells that will not be sampled. The second limitation is that assigning the minimum size to the unsampled cells might result in infeasible timing.

Our current work is not addressing the aforementioned limitations which might, in turn, result in more number of backtracks, although our simulation results are promising. In practice, slew constraints might get provided per cell to ensure that a feasible delay will be attainable. We can then extend our framework to assign the cell size that results in a minimum cost under the provided slew constraint per cell. This gives the minimum feasible cost corresponding to the unsampled cells, and we can still apply our proposed procedure to limit the sampling to a small neighborhood to reduce the number of collected samples.

In this paper, we assign a uniform probability to sample difference sizes of a cell. Furthermore, during sampling of the CR, we assume sampling of all the sizes with uniform probability. However in our simulations, we show near-optimal results are attainable when sampling only two sizes per cell, which are the two discrete sizes around the solution generated using continuous sizing. Using this observation, we can enhance our sampling procedure by implementing a weighted sampling scheme. Our simulation results show that, in the optimal cell sizing solution, most of the cells will have a size which is close to the size generated by continuous sizing. We can incorporate this observation and develop a weighted sampling scheme during which we define a higher probability for a cell to be assigned a size that is close to the size of continuous sizing. In this scheme, the remaining sizes will still have a positive but lower probability to be used during sampling. This helps achieving more useful and feasible samples and, in effect, reduces the number of samples.

Finally, in our implementation of cell sizing, we do not make any assumption on the available sizes, their number, or distance from each other per cell. In applications such as sizing at the post-placement stage, limited space might be available to obtain a legal size per cell. In such case, we can first decide on a range of acceptable sizes per cell based on the current placement solution. This can be done, for example, by examining the amount of white space within each row of a standard cell. We can then apply PaRS for the filtered set of identified sizes.

V. Branch-and-Bound for Cell Sizing

We use BB to find the optimal solution for cell sizing. This allows us to later assess the quality of the solution using nested partitions with respect to the optimal one. Given the potentially huge solution space of BB, it might not be feasible (with respect to runtime) to generate the optimal solution at all. Here, we introduce effective (cell sizing dependent) rules for variable ordering during BB to achieve more effective pruning of the infeasible/suboptimal solution space. Later, we will also discuss how our BB can be implemented on a large-scale grid and run in parallel. These two components (i.e., custom variable ordering and parallel implementation) are the main reasons we can generate the optimal solution using BB in our simulations. In this section, we first give a brief overview of BB for cell sizing and then explain our custom implementation of BB.

A. BB for Cell Sizing

To apply BB, given a level i of the search tree, we compute a lower and an upper bound for each node at level i. We then prune the inferior nodes using the lower and upper bounds. At each node of BB, we have assignment of size variables

for some of the cells and some unassigned variables for the remaining cells. We solve continuous sizing on the unassigned cells to find a lower bound (LB) on that node in the BB tree. Continuous sizing is done by writing a formulation for the unassigned variables given the values of the assigned cell sizes (see Section II). We solve continuous sizing using MOSEK in this paper [19]. We also use the solution of nested partitions as an upper bound for all the nodes in the BB tree. This upper bound (UB) is not only a feasible solution (which is a required condition) but also a very good one (as we show is near optimal). This helps BB to generate the optimal solution faster. We compare these with the current bounds found so far and determine if the node can be pruned out or if the current (lower) bound can get updated to a tighter value. If the node is not pruned out, we further branch on the node.

Furthermore, details about general BB can be found in [20]. In our implementation for cell sizing, we first apply exhaustive search until we reach a level L, after which we apply BB. We did not obtain any effective pruning at the prior levels using BB. This is because, if level L is small, we do not have enough variation of cell size combinations to generate the effective UBs and LBs that can result in pruning of the inferior parts of the solution space.

B. Customized Rules for Effective Pruning

Similar to BB, we always select k cells and explore all their size combinations at each level i in the BB search tree. For example, in Fig. 2, we select variables w_1–w_5 at first. Here, we give two rules on the selection of the variables at each level of the BB tree, which are listed in the following:

1) *Longest Path Rule (LPR):* Here, we choose the k variables that are similar to the partitioning step of PaRS given in algorithm 1. We set all the unprocessed size variables to a minimum size and identify the most timing critical path Pl in the circuit. We then select the top k variables with the highest CF on this path. In the consecutive iteration, we select the next k variables from the same path Pl until all the variables on Pl are processed. This "depthwise" variable ordering helps identifying the large infeasible portions of the search space faster. Note that this variable selection scheme is not exactly the same as in the partitioning step of PaRS in which we dynamically update and select the critical paths at each iteration;
2) *Nearest Neighbor Rule (NNR):* After all the cells on Pl are selected, we choose the cells that are "neighbors" of Pl which are $\{\forall w_l | fanin(l) \in Pl,\ l \notin Pl\}$. The reason is because, at this point, we already explored the feasible assignments on Pl assuming their neighbors have minimum size. Increasing the size of neighbors further makes many of those assignments to become infeasible and allows significant pruning. Once such neighbors are explored, we apply the LPR to find another critical path. *We tried other pruning rules that are based on breadthwise ordering of size variables but did not find them useful for the pruning of the infeasible regions*.

Table I shows the evaluations of our rules on the largest benchmarks. The results are similar for smaller benchmarks.

TABLE I
EFFECTIVENESS OF OUR PRUNING RULES IN EXHAUSTIVE SEARCH: TS (TOTAL SEARCH SIZE), SS (OUR SEARCH SIZE)

Bench	TS	SS	LPR ($\frac{r_1}{q^k}\%$)	NNR ($\frac{r_2}{q^k}\%$)	Level
c5315	> E+230	E+10	0.000027	0.000000	35.11
c7552	> E+230	E+08	0.000007	0.000000	26.93
c6288	> E+230	E+11	0.000002	0.000000	33.98
s1488	E+230	E+11	0.000302	0.000138	32.19
s1494	E+227	E+09	0.000149	0.000116	30.23
s9234	> E+230	E+07	0.000329	0.000057	18.77
s5378	> E+230	E+09	0.000748	0.000587	29.39
s38584	> E+230	E+09	0.000000	0.000000	47.94
s35932	> E+230	E+10	0.000000	0.000000	59.17
b20	> E+230	E+12	0.000000	0.000000	68.34

We applied them to an exhaustive search algorithm and did not apply BB. Columns 2 and 3 give the total search size (TS) and our search size (SS) when LPR and NNR are applied. When choosing $k = 5$ variables using LPR and NNR, out of the q^k possibilities, we found only r_1 and r_2 to be feasible at each node for LPR and NNR, respectively. The average percentages of $(r_1/q^k)\%$ and $(r_2/q^k)\%$ are given in columns 4 and 5 for $q = 2$ and $k = 5$. As can be seen, these two rules for variable selection allow eliminating a large portion of the q^k assignments at each branch on average. Column 6 reports the average of the levels corresponding to the times when processing stops at the branches in the BB tree. For one branch, processing stops when the optimal solution is found at that branch or if no solution is found at all. After processing the cells using LPR and NNR, we apply BB on the remaining cells.

C. Efficient Checking for the Best Objective Value

At each node in the search tree, we initially check if we can obtain a feasible solution when we set the unprocessed sizes to a minimum size. If we find such minimum-cost feasible solution, we do not branch on that node further. This is based on our power models in which the minimum size assignment also corresponds to the minimum cost.

VI. COMPARISON WITH A RECENT SIZING APPROACH

Another recent work on cell sizing is [5]. The algorithm in [5] can be categorized into three steps. A circuit graph is first partitioned into stages by performing breadth-first traversal, and these stages are processed in breadth-first order from the primary input. Inside a stage, a dynamic programming algorithm is applied to search the solution systematically, and a pruning stage is performed to remove inferior solutions. Note that this three-step algorithm is a sequential approach and is very difficult to be parallelized; one stage can be processed only when the stage immediately before it is finished. On the other hand, our customized nested partition approach can fully be parallelized; at one iteration, the current PR is partitioned into k subregions, and these subregions can be processed independently. Furthermore, our sampling procedure provides another level of parallelism. Therefore, we would like to further highlight the importance of having an embarrassingly parallel framework. We provide more discussion on comparison with [5] in our simulations.

Furthermore, as we verified by simulation, there is a high correlation between the sizes generated from continuous sizing and the optimal discrete cell sizes. This assumption has also been utilized in [5]. In such case, we consider only two sizes for each variable, which are the ones that are immediately before and after the solution obtained from continuous sizing in the library for that cell. We can apply BB-2 for our customized BB using two variables. For PaRS, we also considered the variation with two sizes. We modified our algorithms such that, during partitioning and sampling of the subregions, we assumed two sizes for each variable. However, during sampling of the CR, we sampled all the sizes.

VII. Parallel Implementation

We now discuss the parallel implementation of PaRS and BB on a grid of heterogeneous CPUs. We first explain our framework for managing the machines in a heterogeneous grid. We then discuss specific considerations for the efficient parallel implementation of BB and PaRS.

A. Condor System for High-Throughput Computing

Condor system [9] manages the collection of machines in a network to achieve high-throughput computing. The default policy is to assign jobs to CPUs in the network while their owners are away. Consequently, a large computational power is available by exploiting these otherwise-unused cycles without interrupting the machine owners. One Condor feature known as "flocking" allows jobs from a Condor pool to get transferred to another pool, perhaps in another geographical location. Another feature is "glide-in" which allows new computational resources to join a Condor pool. These features promote large-scale (and green) computing.

B. MW-Enabling Grid-Based Optimization

MW is a software framework for parallel computing which is built on top of the Condor system [8], [20]. The concept is that we first break up a large computation into several small tasks to be executed in parallel. Then, a "master" processor sends these tasks to the "worker" processors and retrieves the results when the tasks are finished. Note that these tasks cannot communicate with each other directly. Fig. 7 shows the MW framework to maintain a task list and the relationship between master and worker processors. Note that the master processor is owned by us and cannot be interrupted. Together with Condor, MW provides a dynamic and opportunistic framework for the management of computational resources for the implementation of parallel algorithms.

Using the Condor system, the communication between master and worker machines can be implemented by generating "submit files" in specific format and submitting them to Condor for processing. Using the submit files, we can interact with Condor to find the computing resources for the tasks, to handle communication between master and worker, and to reassign tasks if job migration is necessary. Fig. 8 shows the sample submit files. We can generate a master and a worker submit file

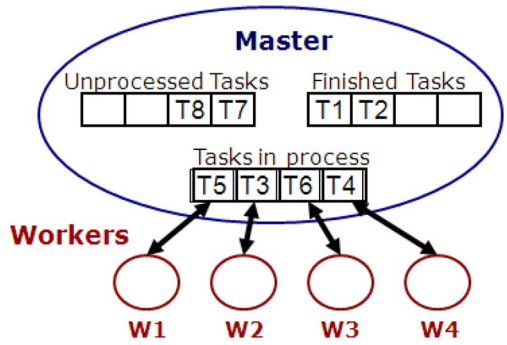

Fig. 7. Using the MW framework, we can dynamically queue the tasks and allocate the computing resources.

MASTER SUBMIT FILE

```
Universe  = Scheduler
Executable = master_DGS_socket
Image_Size = 100000
+MemoryRequirements = 100
Input  = in_master.socket
Output  = out_master.socket
Error  = out_worker.socket
Log  = _DGS.log
Requirements = (Arch == "INTEL" &&
      OPSYS=="LINUX")
getenv  = True
Queue
```

WORKER SUBMIT FILE

```
Universe = Vanilla
#Worker 1
Executable= exec0.$$(Opsys).$$(Arch).exe arguments = 0
8997 8997 144.92.240.35
Log = log_file
Output = output_file.0
Error = error_file.0
Requirements = ( Arch=="INTEL" &&
          OPSYS=="LINUX")
should_transfer_files = Yes when_to_transfer_output =
ON_EXIT
rank = Mips
on_exit_remove = false
Queue
#Worker 2 ...
```

Fig. 8. Example of Condor submit files for master and worker CPUs.

which will be queued under Condor to be processed with other shared jobs in the network. These files include information such as the specification of the target CPUs that should be allocated and the names and arguments of the executable files that should be run by the workers. Instead of using the submit file, we use the C++ application programming interfaces (APIs) of the MW framework which provide the same capability more conveniently.

C. Implementation of Parallel PaRS Using MW

PaRS is an ideal (embarrassingly parallel) optimization algorithm to be implemented using MW. The computation on each subregion can be considered as a task to be done by a worker processor at each iteration. We have q^k workers for q^k subregions and one master processor that receives the information from the workers and decides the next PR. We also have one (or more) workers that apply sampling of the CR.

More specifically, there are two types of workers at an iteration: those assigned to the subregions and those assigned to the CR (note that there can be more than one worker assigned to the CR). A worker performs the following tasks when assigned to a subregion: 1) it receives X_i and Y_i variables indicating the information about the cell sizes corresponding to the current subregion; 2) it performs sampling of the subregion; and 3) it passes on the value of the minimum cost corresponding to the best sample to the master processor. The workers assigned to the CR are each assigned to a few subregions stored in the CR from the previous iterations. In the example in Fig. 6, since seven subregions inside the CR will be sampled at each iteration, we can assign seven workers to take care of sampling

the CR in parallel. Each such worker applies sampling to its assigned subregions and passes the best sample to the master processor. The master processor determines the best sample from the workers in the subregions and CR to find if backtracking is necessary. If not, it determines the index of the next subregion.

D. Implementation of Parallel BB Using MW

In parallel BB, the workers compute the upper and lower bounds for the nodes of the search tree. The master receives the bound information from the workers and applies pruning to decide the next set of workers. The master also applies custom variable ordering that we discussed before, assigning tasks to workers. The main issue in the parallel implementation of BB is the exponential increase in the number of nodes in the search tree in two consecutive levels [8]. Here, the master initially allocates as many workers as possible and assigns tasks to them. When moving to a new level at the search tree, if the number of workers is not sufficient, the master maintains a list of current tasks and their corresponding workers. It also maintains a list of tasks to be assigned upon the availability of resources. If a current task is terminated and its worker becomes free or if a new resource becomes available in the network, the master allocates this as a worker and assigns a new task from the wait list. Fig. 7 illustrates this process. Using MW, we can dynamically assign workers in an opportunistic manner.

E. Communication Overhead

There are three major types of data exchange between the master and a worker: 1) scalar bounds; 2) the variables to be processed; and 3) the netlist information, delay, slew, and leakage models. The latter is the only significant source of overhead.

In current implementation, we send all the information to the worker once it is allocated but reuse each worker as much as possible. Therefore, if a worker will later on process another subregion, it still can use the same information, and no more data communication will be necessary.

In addition, every time a worker is used for a new task, we send to it the new variable assignments. This communication process can get accelerated if the master computes the difference between an old and a new variable assignment that should be sent to a worker and only sends an update.

VIII. Simulation Results

We used a standard-cell 90-nm library provided by TSMC library and initially obtained convex cell delay and leakage power models using curve fitting that is similar to that in [5], [12], and [21]. For details on convex modeling, please refer to [12] and [21]. For convex fitting, we used the nonlinear regression tool of [22]. Convex delay/power models are necessary in continuous sizing which can be used to bias the search. Our cost was a linear combination of area and power [see (1)]. The number of sizes per cell in the library varied between three to ten and, on an average, was 4.48. We implemented the following approaches:

1) **LB:** This is continuous sizing assuming each cell size i is a continuous quantity within $S_i^{\min}$ and $S_i^{\max}$ provided by the library. We use the continuous sizing solution as an LB for minimum attainable cost. LB is solved as a convex program using MOSEK (see Section II). We used the APIs provided by MOSEK in conjunction with our C++ code;
2) **NR:** This is the nearest rounding (NR) approach. We round the solution of LB to the closest size. This solution violates the timing. We then upsize the cells one by one based on their CF [see (2)] until the delay is satisfied. We update the CFs of all the cells every time a cell is upsized;
3) **GR1:** This is a gradient-based approach. First, all the cells are set to a minimum size. Then, they are upsized one at a time. Upsizing is based on the CF using the current size and the cost of the immediately larger size for the cells. After each upsizing, one incremental timing analysis is done to update the CF values;
4) **GR5:** This is similar to GR1, only the timing analysis and update of the CF values are done after every five upsizings;
5) **PSA:** This is parallel simulated annealing. We run the variations of simulated annealing implementation with different random seeds on 20 CPUs. In our simulated annealing implementation, we start from the solution of continuous sizing and use GR5 to quickly get a feasible solution as the starting point. Then, at each iteration of simulated annealing, we apply a random downsizing (for cost improvement) followed by a random upsizing (to ensure getting a feasible solution). During downsizing, we go through the cells exactly once on the order of their CFs and randomly decide if each should be downsized. We then go through the cells on the order of their CF again and randomly upsize them until the timing constraint is satisfied. We also make the probability of downsizing (for cost improvement) higher for the higher temperatures;
6) **BB-ALL:** This is parallel BB using all the discrete cell sizes in the library which generates the optimal cell sizes;
7) **BB-2:** We also apply BB assuming each cell has only two sizes (BB-2) which are the ones immediately before and after the solution of LB. Refer to Section VI for more details. We verified via simulation that the percentage difference between the sizes generated by BB-ALL and BB-2 for each cell is low;
8) **PaRS** (our work): We selected two as described for BB-2. We chose $k=4$ variables which are the number of selected cells for partitioning at each iteration. We also assumed two sizes per cell similar to BB-2, resulting in 16 subregions at each iteration. We collected $M = 100$ samples at each iteration for the subregions and for the CR. The subregions were processed in parallel, but, in our implementation, the sampling step was done serially.

Parallel implementations of BB-ALL, BB-2, and PaRS were done using Condor and MW software on the grid of Computer-Aided Engineering at the University of Wisconsin—Madison.

TABLE II
COMPARISON OF THE TOTAL COST. THE REPORTED NUMBERS ARE THE PERCENTAGE DISTANCE FROM THE OPTIMAL SOLUTION BB-ALL

Bench	#cells	BB-2	NR	GR5	GR1	PSA	PaRS
c432	136	0.00	7.99	10.30	6.47	0.44	0.00
c499	163	0.30	18.37	19.31	14.07	3.92	2.11
c880	208	0.04	12.66	13.36	11.31	7.44	0.04
c1355	205	0.00	2.66	6.77	4.46	3.26	0.00
c1908	199	0.15	27.63	26.52	24.19	13.13	0.66
c2670	472	0.06	11.30	12.23	9.76	7.04	0.70
c3540	479	0.03	17.17	23.74	20.39	11.96	1.65
c5315	705	0.37	3.34	11.77	10.68	7.60	1.02
c6288	1256	0.16	2.13	9.26	7.88	4.34	0.62
c7552	822	0.00	4.28	2.28	2.13	1.14	0.09
s1488	307	0.00	13.41	15.23	9.10	4.76	2.33
s1494	309	0.00	8.85	12.44	6.83	6.09	1.83
s5378	930	0.07	7.09	4.77	3.07	2.09	0.14
s9234	740	0.14	50.60	54.91	34.59	27.62	0.14
s15850	617	0.06	10.19	11.93	6.21	4.06	0.53
s35932	7260	0.00	14.79	18.79	14.04	8.29	0.00
s38584	6950	0.00	2.93	3.66	3.20	2.48	0.08
b18	47191	0.17	13.12	13.52	9.80	7.19	0.84
b20	15699	0.11	11.40	11.57	11.32	9.09	1.02
b22	24484	0.09	1.29	4.65	4.22	3.26	0.41
Ave.		0.09	12.63	14.86	10.69	6.76	0.70

TABLE III
COMPARISON OF RUNTIME (IN SECONDS)

Bench	BB-All	BB-2	GR5	GR1	PSA	PaRS	#BT
c432	51	22	1	1	22	1	0
c499	390	120	1	1	31	4	1
c880	63	21	1	1	33	2	0
c1355	60	45	1	1	28	4	1
c1908	6844	111	1	1	51	6	2
c2670	1258	45	1	1	38	8	2
c3540	78240	11380	1	1	81	17	1
c5315	2156	581	1	1	123	21	5
c6288	3992	1239	1	1	149	45	0
c7552	1870	682	1	1	127	14	3
s1488	125671	88	1	1	42	8	1
s1494	3233	197	1	1	39	5	2
s5378	2332	133	1	1	131	19	2
s9234	3105	341	1	1	77	9	1
s15850	2872	420	1	1	83	7	1
s35932	3174	159	52	69	294	142	0
s38584	3700	576	10	23	462	369	4
b18	188862	41721	1997	27341	9931	1117	1
b20	103384	19528	235	5004	1272	613	3
b22	138473	23112	458	9251	1409	681	6

The site featured 179 Intel/Linux CPUs, and it flocked to the site of Computer Science in case no CPU was free. The master machine had a 2.2-GHz processor with a 2-GB memory running Linux. Our program and MW APIs were in C++. We used ISCAS85', '89, and '99 combinational and sequential benchmarks which were synthesized using Synopsys Design Compiler.

The timing constraint in our simulations was decided as follows. We first find T, the circuit timing of the case when all the cells have a minimum size. We defined the timing constraint in our simulations empirically as $0.75T$. We found out that any smaller timing constraint made the continuous sizing solver not to generate a feasible solution for most of the benchmarks. Our timing constraint was, therefore, very stringent.

Comparison of the Total Cost: Table II reports the total cost of each approach. Columns 3–8 list the percentage difference between BB-2, NR, GR5, GR1, PSA, and PaRS from the optimal one (BB-ALL). On average, NR, GR5, GR1, and PSA are 12.6%, 14.86%, 10.69%, and 6.76% away from the optimal one, while we are within 1% of the optimal one. BB-2 also generates near-optimal solutions. All timing constraints were met.

Note that NR and GR5, GR1, and PSA all behave inconsistently over the benchmarks. Over the 20 benchmarks, in ten cases, the solutions of NR, GR5, and GR1 were all more than 9% away from the optimal one. For S9234, these three approaches were at least 34.6% away from the optimal one, which is very significant. Furthermore, PSA was also away from the optimal one by more than 9% in four benchmarks (c1908, c3540, c9234, and b20). On the other hand, the maximum error of PaRS was only 2.33% over all the 20 benchmarks. We therefore conclude that PaRS *robustly* generates a near-optimal solution for a variety of benchmarks, while the other approaches all perform inconsistently.

In GR5, GR1, and PSA, part of the reason for suboptimality is the use of CF which depends on the change in the cell delay per change in its cost. The change in a cell delay might not necessarily translate in the same amount of change in the circuit delay, leading to an erroneous solution. Alternatively, if the CF was defined based on the change in the circuit delay, a separate timing analysis would have been needed per cell for the estimation of each CF value which would have made the runtime of the algorithm impractical.

Please note that, in our simulations, achieving a near-optimal solution quality by using only two sizes might be a function of our chosen library sizes, types of gates used during synthesis, and other implementation factors.

Comparison of Runtime and Number of Workers: Table III reports the (wall clock) execution runtimes. The unit of runtime is seconds, and the runtimes include all the communication overhead. PaRS is done in seconds, whereas BB-2 and BB-ALL are done in hours or even days. The average runtime improvements of PaRS over BB-ALL and BB-2 were 1211X and 56X, respectively. Compared to the other approaches, for the first 15 benchmarks that were smaller in size, GR1 and GR5 were done in 1 s, while the maximum runtime of PaRS was 50 s (still a small number). Overall, PSA was the slowest compared to GR1, GR5, and PaRS and had a maximum runtime of 149 s for the first 15 benchmarks. For the final five benchmarks which were the largest, PaRS has a comparable runtime to GR1 and for b18, it even achieved a 22X runtime improvement (27341 s versus 1117 s). PaRS was still slower than GR5, but the solution quality of GR5 was significantly worse than PaRS (and of GR1). The runtime gap between PaRS and PSA closes for the last five benchmarks, and PSA becomes faster than GR1. Overall, PaRS consistently has better runtime than PSA. Column 8 reports the number of backtrackings in PaRS which varied between 0 and 11. It was, on average, 2.3 over the 20 benchmarks.

Table IV also reports the total number of worker CPUs for the parallel approaches. We report both the maximum and average number of workers. The maximum corresponds to the total number of allocated workers while the program was running. The average corresponds to the average number of workers over the runtime of the program. The maximum number of workers in BB-ALL and BB-2 were 192 and 187, respectively. PaRS used only 20 workers in which 16 workers were allocated to

TABLE IV
COMPARISON OF THE NUMBER OF CPUS AND RUNTIME IN THE PARALLEL TECHNIQUES; THE MAXIMUM NUMBER OF CPUS IN PaRS AND PSA IS 20

	Number of CPUs				Runtime × Average CPUs (×0.001)			
	BB-ALL		BB-2		BB-ALL	BB-2	PSA	PaRS
Bench	Max	Ave	Max	Ave				
c432	81	75.5	60	50.1	4	1	0.44	0.02
c499	134	56.7	83	81.1	20	10	0.62	0.08
c880	64	42.7	71	60.1	3	1	0.66	0.04
c1355	94	80.3	65	60.1	5	3	0.56	0.08
c1908	189	142.0	102	99.8	1000	10	1.02	0.10
c2670	77	57.5	82	71.5	70	3	0.76	0.20
c3540	162	157.8	130	125.3	10000	1000	1.62	0.30
c5315	118	105.7	73	53.6	200	30	2.46	0.40
c6288	126	114.9	101	92.4	500	100	2.98	0.90
c7552	113	102.0	133	127.2	200	90	2.54	0.30
s1488	156	114.8	82	76.8	10000	7	0.84	0.20
s1494	130	111.8	89	85.4	400	20	0.78	0.10
s5378	129	95.6	89	78.5	200	10	2.62	0.40
s9234	119	95.8	102	98.0	300	30	1.54	0.20
s15850	139	112.7	118	101.4	300	40	1.66	0.10
s35932	163	108.2	94	79.0	300	10	5.88	3.00
s38584	133	113.9	108	94.5	400	50	9.24	9.00
b18	192	189.8	187	179.9	40000	8000	199	20.0
b20	187	167.3	143	124.3	20000	2000	25.40	10.0
b22	190	173.7	155	128.3	20000	3000	28.20	10.0

16 subregions, along with four workers that were assigned to the CR. In C432 and C880, the number of workers in PaRS were 16 and 19, respectively. This is because of the small size of these benchmarks; the program terminates before all the workers of the CR are allocated on the grid. Furthermore, PSA had 20 worker CPUs which were instantiated with different random seeds.

Table IV also reports the multiplication of runtime T and the average number of workers W for BB-ALL, BB-2, PSA, and PaRS. As can be seen, PaRS provides the best value in terms of both runtime and number of CPUs.

Considering this with the solution quality and runtime numbers, we conclude that, overall, PaRS reaches the best tradeoff in terms of solution quality (gets near-optimal quality) and runtime (better than all the parallel approaches in all the benchmarks and the sequential approaches for larger benchmarks) while considering the number of CPUs. *We would like to note that, at each iteration of PaRS, we consider many subregions but only select one (while aggregating the rest in the CR). We do not consider this to be a "waste" of computing resources because, overall, we get better solution quality and faster runtime with a small number of CPUs.*

Comparison of Runtime Scalability: Fig. 9 shows the runtime plots as a function of the number of cells over the benchmarks (except S1488 and C3540). As can be seen, PaRS is the most scalable.

Note that the plots are all for parallel implementation. Without parallel implementation and dynamic resource allocation, the scalability of BB-2 and BB-ALL was significantly worse.

Compared to [4], we had better runtime for all the benchmarks; however, it is not fair to compare the runtime since we used different library and had different cost function (area/power). Instead, we would like to emphasize the runtime scalability. The runtime of C432 and C7552 reported in [4] are 31 and 3083 s, respectively, and the runtime increment of C7552 over C432 is about 99. On the other hand, the runtime

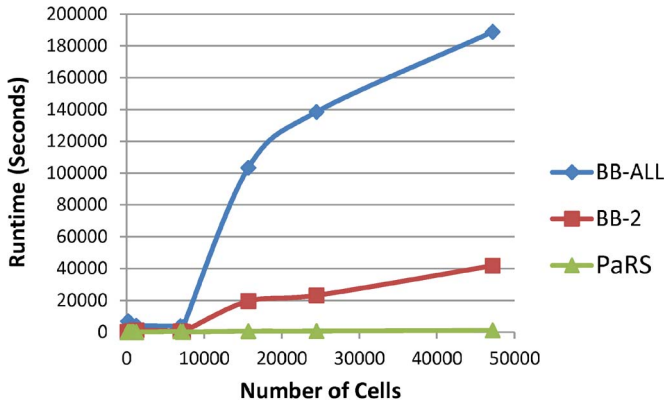

Fig. 9. Scalability comparison of different approaches.

increment of these two benchmarks in our approach is only 14. This is an example to show that our approach can potentially provide better runtime scalability.

Effect of Varying the Number of Workers in PaRS: In the implementation of PaRS, we set $k = 4$, so we chose four cells at each iteration for sizing and had 16 subregions for two cell sizes. This value of k was empirically set based on our benchmarks. We observed not to have too many combinational logic levels for the longest paths in our benchmarks. The value of $k = 4$ was covering about 25%–30% of the length of the longest path at each iteration.

We also experimented with increasing k to five and six. Fig. 10 shows the total cost obtained for the three values of k. These costs are normalized to the cost corresponding to $k = 4$. These costs are shown for the last ten benchmarks in Table II, and the behavior over the remaining benchmarks was similar. As can be seen, a slight decrease in cost can be achieved by increasing k. However, this decrease is within 2% and, in most cases, within 0.5% of $k = 4$. Therefore, our choice of k is more desirable as it yields to fewest number of worker computers.

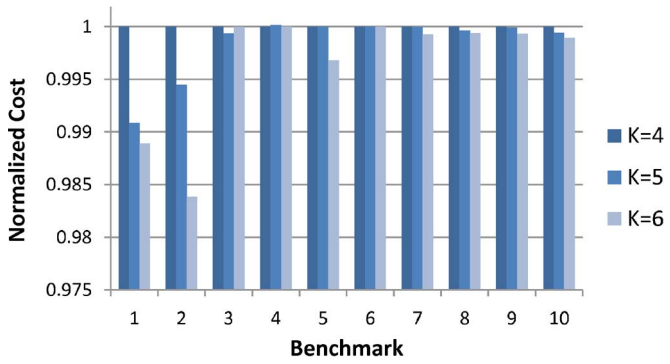

Fig. 10. Impact of changing the number of workers on the total cost.

IX. Conclusion and Future Work

We have presented PaRS, a parallel and randomized algorithm for cell sizing. It is scalable and achieves a near-optimal solution within seconds for benchmarks of varying sizes and topologies using only 20 CPUs. We have also presented a parallel and custom BB algorithm for cell sizing which uses up to 200 CPUs to find the optimal solution. Overall, the combination of partitioning and random sampling of the solution space in PaRS is the key in effectively identifying the promising search direction to the optimal solution, thereby achieving significant speedups without losing optimality using only 20 CPUs. We feel that formulating nested partitions is a suitable solution for parallel CAD which can work with a reasonable number of CPUs. Furthermore, nested partitions is a hybrid framework which allows integration of other heuristic and formal approaches to cell sizing. This, again, is a nice feature for parallel CAD which is based on the reuse of existing algorithms.

In the future, we plan to find parallel solutions to other CAD problems and also explore a heterogeneous grid in which each grid node could be a multiprocessor. We would also like to consider more interesting extensions such as combined sizing with buffer insertion, as, often, a suboptimal sizing with buffering might be a more effective approach to solely sizing to fix timing violations. We can extend our nested partitions implementation to add new variables for buffering, and the current work is the first step toward that direction.

References

[1] T.-H. Wu and A. Davoodi, "PaRS: Fast and near-optimal grid-based cell sizing for library-based design," in *Proc. Int. Conf. Comput.-Aided Des.*, 2008, pp. 107–111.

[2] S. Sapatnekar, V. Rao, P. Vaidya, and S. Kang, "An exact solution to the transistor sizing problem for CMOS circuits using convex optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 11, pp. 1621–1634, Nov. 1993.

[3] C.-C. Chen, C. Chu, and D. Wong, "Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation," in *Proc. Int. Conf. Comput.-Aided Des.*, 1998, pp. 617–624.

[4] F. Beeftink, P. Kudva, D. Kung, and L. Stok, "Gate size selection for standard cell libraries," in *Proc. Int. Conf. Comput.-Aided Des.*, 1998, pp. 545–550.

[5] S. Hu, M. Ketkar, and J. Hu, "Gate sizing for cell library-based designs," in *Proc. Des. Autom. Conf.*, 2007, pp. 847–852.

[6] O. Coudert, "Gate sizing for constrained delay/power/area optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 5, no. 4, pp. 465–472, Dec. 1997.

[7] L. Shi and S. Ólafsson, "Nested partitions method for global optimization," *Oper. Res.*, vol. 48, no. 3, pp. 390–407, May 2000.

[8] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "Master–worker: An enabling framework for applications on the computational grid," *Cluster Comput.*, vol. 4, no. 1, pp. 63–70, 2001.

[9] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, *Condor—A Distributed Job Scheduler*, T. Sterling, Ed. Cambridge, MA: MIT Press, Oct. 2001.

[10] D. Nguyen, A. Davare, M. Orshansky, D. Chinnery, B. Thompson, and K. Keutzer, "Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization," in *Proc. Int. Symp. Low Power Electron. Des.*, 2003, pp. 158–163.

[11] J. Wang, D. Das, and H. Zhou, "Gate sizing by Lagrangian relaxation revisited," in *Proc. Int. Conf. Comput.-Aided Des.*, 2007, pp. 111–118.

[12] K. Kasamestty, M. Ketkar, and S. Sapatnekar, "A new class of convex functions for delay modeling and their application to the transistor sizing problem," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 7, pp. 779–788, Jul. 2000.

[13] H. Tennakoon and C. Sechen, "Efficient and accurate gate sizing with piecewise convex delay models," in *Proc. Des. Autom. Conf.*, 2005, pp. 807–812.

[14] L. Shi, S. Ólafsson, and N. Sun, "A new parallel and randomized algorithm for the traveling salesman problem," *Comput. Oper. Res.*, vol. 26, no. 4, pp. 371–394, 2000.

[15] L. Shi, S. Ólafsson, and Q. Chen, "A new hybrid optimization algorithm," *Comput. Ind. Eng.*, vol. 36, no. 2, pp. 409–426, Apr. 1999.

[16] L. Shi, S. Ólafsson, and Q. Chen, "An optimization framework for product design," *Manage. Sci.*, vol. 47, no. 12, pp. 1681–1692, Dec. 2001.

[17] L. Shi and S. Ólafsson, "Convergence rate of the nested partitions method for stochastic optimization," *Methodol. Comput. Appl. Probab.*, vol. 2, no. 1, pp. 37–58, 2000.

[18] J. Fishburn and A. E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *Proc. ICCAD*, Nov. 1985, pp. 326–328.

[19] *MOSEK Optimization Software*. [Online]. Available: http://mosek.com/

[20] J. T. Linderoth, "Topics in parallel integer optimization," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, 1998.

[21] H. Chou, Y.-H. Wang, and C.-C. Chen, "Fast and effective gate-sizing with multiple-Vt assignment using generalized Lagrangian relaxation," in *Proc. Asia South-Pacific Des. Autom. Conf.*, 2005, pp. 381–386.

[22] *Nonlinear Regression Analysis Program*. [Online]. Available: http://www.nlreg.com/

Tai-Hsuan Wu (S'08) received the B.S. degree from the Department of Electrical and Control Engineering, National Chiao Tung University, Hsinchu City, Taiwan, in 2002. He is currently working toward the Ph.D. degree in the Department of Electrical and Computer Engineering, University of Wisconsin, Madison.

His research interests are in the area of computer-aided design and with emphasis on power aware and high-performance design techniques, parallel optimization, and grid computing environments.

Azadeh Davoodi (M'06) received the B.S. degree in electrical and computer engineering from the University of Tehran, Tehran, Iran, in 1999 and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland, College Park, in 2002 and 2006, respectively.

Since 2006, she has been an Assistant Professor with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison. Her research interests are in design automaton in the presence of manufacturing-induced variations, parallel optimization on different computing platforms, and power reduction computer-aided design techniques.