An Efficient Algorithm for Finding Empty Space for Online FPGA Placement

Manish Handa and Ranga Vemuri

{mhanda,ranga}@ececs.uc.edu

Department of ECECS, University of Cincinnati
Cincinnati, OH 45221-0030, USA

ABSTRACT

A fast and efficient algorithm for finding empty area is necessary for online placement, task relocation and defragmentation on a partially reconfigurable FPGA. We present an algorithm that finds empty area as a list of overlapping maximal rectangles. Using an innovative representation of the FPGA, we are able to predict possible locations of the maximal empty rectangles. Worst-case time complexity of our algorithm is $\mathcal{O}(xy)$ where x is the number of columns, y is the number of rows and $x.y$ is the total number of cells on the FPGA. Experiments show that, in practice, our algorithm needs to scan less than 15% of the FPGA cells to make a list of all maximal empty rectangles.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Design Aids—*Placement and Routing*; J.6 [**Computer-aided Engineering**]: Computer-aided Design(CAD)

General Terms

Algorithm Design

Keywords

Online Placement, Partially Reconfigurable FPGAs, Reconfigurable Computing

1. INTRODUCTION

Partially reconfigurable FPGAs (Field Programmable Gate Arrays) are capable of modifying their logic and interconnect configuration at run-time to efficiently execute the application at hand. In such systems, an application is divided into smaller tasks that run concurrently on the FPGA. After a task finishes execution, it is removed from the FPGA and the area can be reclaimed and reused by next set of tasks.

In the online scenario, the flow of execution of the applications is not known in advance. The sequence of tasks needed for each application is known only at run time. Placement in such an environment is called *online placement*. The placement time is an overhead on total execution time of the application. Finding and managing empty space on the FPGA is a time consuming part of a placement algorithm. So a fast algorithm is required to efficiently manage empty space for fast placement of tasks.

Fragmentation of the FPGA resources is known to cause low area utilization [1, 2] in the dynamic reconfigurable systems. Defragmentation and task relocation need an efficient algorithm to find empty space on the FPGA. In addition, incremental placement algorithms also need an efficient way to find empty area on the rectangular surface to place the perturbed circuit elements.

Empty space on the surface of a FPGA, with some rectangular tasks placed on it, can be covered using a finite set of rectangles called empty rectangles.

Definition A *Maximal Empty Rectangle (MER)* is the empty rectangle that cannot be covered fully by any other empty rectangle.

In this paper, we propose a fast algorithm for finding empty space in a partially reconfigurable FPGA. The empty space is maintained as a list of overlapping maximal empty rectangles. A placement algorithm can use our algorithm to find a suitable placement location for a task on the FPGA. In addition, our algorithm can be used to refresh the empty area after the addition or deletion of a task.

Rest of the paper is organized as follows: Section 2 cites some related work. In Section 3, we explain the motivation for maintaining the empty area as a list of maximal empty rectangles. In Section 4, we give details of our algorithm for finding a list of MERs. Section 5 gives time and space complexity of our algorithm and in Section 6, we provide some experimental results.

2. RELATED WORK

Finding and maintaining empty space on a surface is a fundamental problem in computational geometry [3, 4]. Edmonds et al. proposed an algorithm for searching empty space in large two-dimensional data-sets [5]. In our work, we make significant improvements upon the average run time of the algorithm proposed in [5].

Bazargan et al. [6] proposed heuristics for maintaining empty area in the form of non-overlapping rectangles. They sacrificed quality of placement for attaining high speed. In our opinion, quality of placement is as important as speed for online placement algorithms. Walder et al. [7] improved

DAC 2004, June 7–11, 2004, San Diego, California, USA.

the quality of placement by delaying free space partitioning. Partitioning of free space is not optimal In their heuristic. Bazargan et al. showed that maintaining empty space as maximal rectangles leads to better utilization of resources. We propose a fast algorithm for finding maximal rectangles. Our algorithm produces better quality results than [6, 7] because we maintain maximal rectangles.

3. MOTIVATION FOR MAINTAINING MAXIMAL EMPTY RECTANGLES

The heuristics proposed in [6] maintain empty area as a list of non-overlapping rectangles (as opposed to overlapping maximal empty rectangles). As a result, the empty rectangles may not be maximal and cannot be used for placement of a slightly larger rectangle which could be placed had maximal rectangles been maintained. Figure 1 shows one such instance. A task T1 is added to the FPGA area

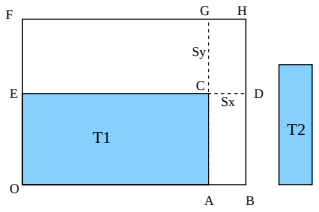

Figure 1: ***Maximal Vs Non Maximal Empty Rectangles***

OBHF. This results in two overlapping maximal rectangles - ABHG and EDHF. If we keep only non-overlapping rectangles and divide empty space across segment S_x, it gives two non-overlapping rectangles ABDC and EDHF. Another task T2 need to be place on the FPGA. Since the heuristic in [6] maintains only non-overlapping, non-maximal empty rectangles, the task T2 cannot be placed on the FPGA although there is sufficient area on the FPGA for placement of T2. Non-optimality of this heuristic will result in more fragmentation on the FPGA and area utilization will reduce. Similar arguments can be made if we choose to divide the empty space along segment S_y.

Maintaining empty area as overlapping MERs leads to 8% better placement quality[6].

We have concluded after a large number of experiments that the number of overlapping maximal rectangles and the number of non-overlapping non-maximal rectangles are nearly the same. Since number of empty rectangles are same in both the cases, the memory requirements for storing them will be same and so will be the time to search through the list of empty rectangles for finding a suitable candidate for task placement. So, storing overlapping maximal rectangles gives better placement quality at no penalty.

4. FINDING MAXIMAL EMPTY RECTANGLES

In this section, we discuss our algorithm for finding all maximal empty rectangles. We start with FPGA modeling technique and the datastructures used in our algorithm.

4.1 FPGA Surface Modeling

In our model of partially reconfigurable FPGA, each application is divided into a set of smaller tasks. Each task is viewed as a hard rectangular task with a finite height and width. Orientation of the task is fixed and we do not allow task rotation during placement (however, such rotation can be contemplated outside the algorithm discussed here). Each task takes finite time for its execution, called its life-time. After a task is executed for its life-time, it is removed from the FPGA and its area can be used to place other task(s). Each task has a arrival time, which is the earliest time a block may be placed on the FPGA. The tasks are not allowed to overlap each other on the FPGA surface.

The FPGA surface is modeled as a two dimensional array with X number of columns and Y number or rows. This array is called *area matrix*. Each cell in the array represents a CLB in the FPGA. Bottom left cell is addressed $(1, 1)$ and top right cell is addressed (X, Y)

If a cell (CLB) in the *area matrix* (FPGA) is used by a task, the cell is called an occupied cell, otherwise it is called an empty cell. Each cell in *area matrix* has a value stored in it. We call this value the weight of the cell. Figure 2 shows the *area matrix* of a FPGA with two tasks placed on it. Each empty cell in the FPGA is represented by a positive number and every occupied cell is represented by a negative number. In addition, a positive number in a cell gives number of contiguous empty cells above and including that cell in that column and a negative number in an occupied cell gives the remaining width of the task measured in the right direction from that cell. This coding of the cells is very important for efficient implementation of our algorithms. Detailed use of coding technique is explained in subsequent sections.

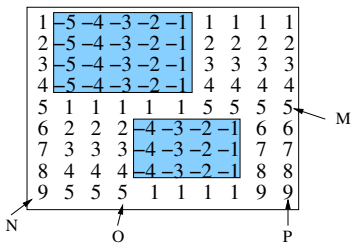

Figure 2: ***Modeling FPGA Area Matrix***

The *area matrix* data structure can be updated very efficiently after addition or deletion of a task. Negative weight is assigned to all the cells occupied by the added task. All the positive cells directly below the task are incremented by height of the newly added task. Similarly, after deletion of a task, positive weight is assigned to all the cells previously occupied by the deleted task. New weights reflect the number of continuous empty cells above the cell.

4.2 Staircase Data Structure

We maintain free area on the FPGA in the form of empty rectangles. All the empty rectangles having same bottom right coordinate make a *staircase*.

Definition A *staircase*(x, y) is defined as the the collection of all empty rectangles with (x, y) as their lowest right virtex. Point (x, y) is called *origin* of the staircase.

Figure 3 shows example of a *staircase*. The *staircase* is made at point $O(x, y)$ and it contains three empty rectangles with their diagonals as OB, OC, and OD respectively.

A *staircase* at a point (x, y) can be fully represented by top leftmost points of each of its stairs. Note that all the weights

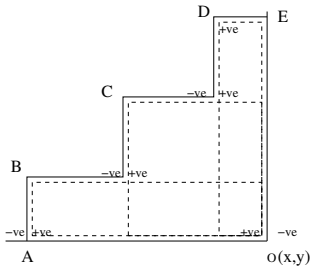

Figure 3: *A Staircase*

inside the *staircase* have to be positive, representing empty space. The negative entries define the boundary and the shape of the *staircase*. More details on *staircase* data structure can be found in [5, 8]. However, [5] uses a completely different encoding based on 0's and 1's. Our new encoding method described above leads to significant speedups as we will see later in this paper. Figure 4 shows some *staircases* in a FPGA in which some tasks have been placed. *Staircase* at point P has three stairs with their top leftmost edges at A, B and C respectively. *Staircase* at point M has only one stair with its top edge at C.

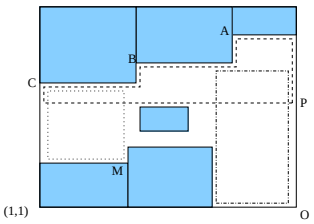

Figure 4: *Example of Staircases on a Area Matrix*

4.2.1 Construction of Staircases

Construction of *staircases* is described in [5]. We are including the construction method here for the purpose of completeness.

At the first leftmost positive entry of each scanned row, there is only one stair in the *staircase* and height of that stair is given by positive integer stored at that location. A *staircase* at point $(x+1, y)$ can be easily constructed from a *staircase* at point (x, y). Figure 5 shows construction of *staircase*$(x+1, y)$ from a *staircase*(x, y), shown in dotted line. Let (x, y') be the top rightmost point on the stair containing column x and $(x+1, y'')$ be the coordinate of topmost empty cell on in the column $x+1$. Construction of *staircases* in three possible cases is shown in Figure 5.

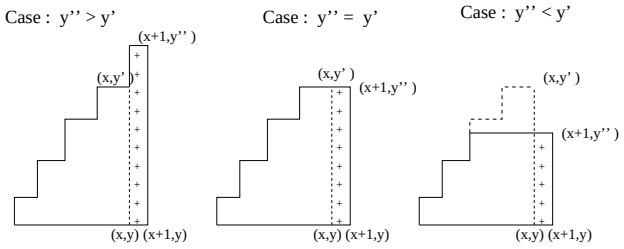

Figure 5: *Construction of a Staircase*

- **Case :** $y'' > y'$A new stair is added to the *staircase*. y" becomes top-left corner of the new stair
- **Case :** $y'' = y'$ Width of the rightmost stair is extended by one.
- **Case :** $y'' < y'$ All the stairs with height greater than that of y" are deleted and the last deleted step is replaced with new highest step. New highest step has height equal to y".

4.3 Extracting Maximal Empty Rectangles from a Staircase

We state and prove the following Theorem.

THEOREM 1. *Any maximal empty rectangle will be contained in one and only one staircase.*

PROOF. First we prove that any maximal rectangle will be contained by at least one *staircase*. Let us assume a maximal empty rectangle R on the *area matrix*. Let $O(x, y)$ be its bottom right coordinate and $E(v, w)$ be its top left coordinate. A *staircase* S can always be made at $O(x, y)$. S will have at least one stair with E as its top-left edge and R will be contained by S.

In addition, a maximal empty rectangle is contained by only one *staircase*. This can be proved by fact that only one staircase can be constructed at any one point and the maximal rectangle R is contained only in the staircase constructed at its bottom right coordinate. So, a maximal rectangle is contained in only one *staircase* that has the origin at its bottom right corner. □

We scan the *area matrix* on a row-by-row basis from top to bottom and make *staircases* at the empty locations. The maximal empty rectangles (MERs) are extracted out of the *staircases*. Note that all the *staircases* do not necessarily contain MERs but, by Theorem 1, each maximal empty rectangle is contained by some *staircase*. In [5], each *staircase* is examined for maximal rectangles. In next section, we propose heuristics for avoiding checking the *staircases* that cannot contain maximal rectangles. We check only those *staircases* that can have maximal empty rectangles in them.

Definition A *maximal staircase* is the *staircase* with at least one maximal empty rectangle in it.

A *staircase* is maximal if it cannot be extended down or to its right. If the vertical edge of the *staircase* can be extended to the right, the new rectangles in the new *staircase* will cover all the old rectangles. Same is true for the bottom edge of the *staircase*. We check only *maximal staircases* for extracting maximal empty rectangles. The question is how to identify *maximal staircases*. Answer to this question is provided in the next section.

After a *maximal staircase* is detected at a point, all the rectangles in the *staircase* can be checked to see if they are maximal empty rectangles. All maximal rectangles are reported and the algorithm proceeds to next point.

Whether the bottom or the right edge of the *staircase*(x, y) can be extended depends upon the location of the occupied cells in column x+1 and row y-1. Let x^* be the X-coordinate of leftmost positive entry of a horizontal block of positive entries starting at $(x, y-1)$. Let y^* be the Y-coordinate of topmost positive entry of a vertical block of positive entries

starting $(x+1, y)$. Consider a step in $staircase(x, y)$ with top left corner (x_i, y_i). The rectangle (x_i, x, y_i, y) is maximal if and only if $x_i < x^*$ and $y_i > y^*$ [5].

4.4 Finding Maximal Staircases

Since occupied locations are inherently clustered in our encoding, we propose the following two theorems for locating *maximal staircases* on the *area matrix*. Note that both of these are necessary conditions for the location of a *maximal staircase*, but not sufficient.

THEOREM 2. *Each maximal* staircase *rests on top horizontal boundary of an already placed task or on the bottom of the area matrix*

PROOF. Let us consider a *maximal staircase* S as shown in Figure 6. Let us assume that the *staircase* does not lie over the top horizontal boundary of any task. T1 is the task nearest to bottom boundary of *staircase*. Let us assume that horizontal boundary of the *staircase* can not be moved in right direction due to a task placed to its right.

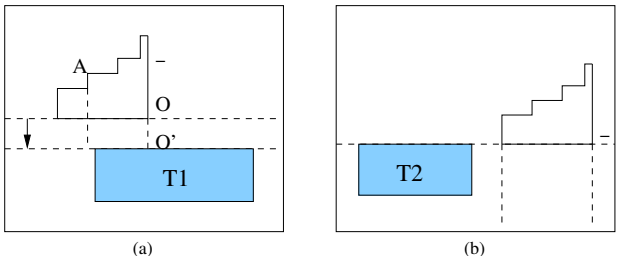

Figure 6: ***Horizontal Boundary of a Maximal Staircase***

Let us consider a rectangle in the *staircase* with OA as one of its diagonals. The rectangle is certainly not a maximal empty rectangle. This is because rectangle O'A covers that rectangle fully. The same is true for each rectangle in the *staircase*. So, S does not contain any maximal rectangle and so, it can not be a maximal staircase.

If the *staircase* is extended down till it touches boundary of the task T1, It will have O'A as a maximal empty rectangle. So, only a *staircase* at the top horizontal boundary of an already placed task can be a *maximal staircase*.

These observations can be extended to include the bottom row of the area matrix as a possible location for the bottom boundary of a *maximal staircase*. □

The *area matrix* in the Figure 6(b) shows insufficiency of the condition proved in Theorem 2. *Staircase* shown in the Figure lies on top of a task but it is not a *maximal staircase*. So, Theorem 2 gives a necessary condition for a staircase to be maximal. This condition is not sufficient.

THEOREM 3. *The origin of each maximal staircase lies on the cell whose weight is more than that of cell on its right or there are no cells to its right.*

PROOF. Let us consider a *staircase* with C as origin on the *area matrix* as shown in Figure 7. Let D be the weight of the cell at the immediate right of C. Let A and B be other points on the lower boundary of the *staircase*. Also assume that the *staircase* lies on top horizontal boundary

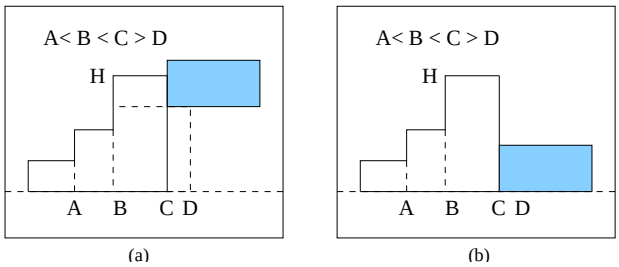

Figure 7: ***Vertical Boundary of Maximal Staircase***

of an already placed task. So, it can not be extended to its bottom side.

If $D \geq C$, the *staircase* can be extended to its right as shown in Figure 5 and newly formed rectangles cover the previous rectangles. So, the *staircase* is not maximal. If $D < C$, then two possibilities exists: either D is positive (Figure 5(a)) or D is negative (Figure 5(b)). If D is negative, then cell D is occupied and *staircase* can not be extended to its right. In that case, the *staircase* can be maximal. If D is positive then, the *staircase* can be extended to its right but height of newly formed *staircase* will be smaller. In that case, rectangle CH can be a maximal rectangle and so the *staircase* can be maximal.

So, the origin of a *maximal staircase* is found at a cell where value in the right cell is less than the value of the cell at the origin.

These arguments can be extended to include right boundary of the area matrix as a possible location for the vertical boundary of a *maximal staircase*. □

4.5 Selective Construction of Staircases

Theorems 2 and 3 can be used to construct only a small fraction of all possible staircases as possible *maximal staircase* candidates. The *area matrix* is scanned to make *staircases* at empty locations. Coding of the *area matrix* helps in its quick scanning.

We scan the *area matrix* on a row-by-row basis from top to bottom. In [5], each location is checked to see if that is empty. In our application, the occupied cells exist in a cluster. We use this fact to improve runtime for our algorithm. We do not scan all the rows. We scan the rows immediately above the top horizontal boundary of already placed tasks and the bottom-most row and skip scanning of all other rows. In each scanned row, scanning is done from the left side to the right side. If a negative integer i is encountered, the number of entries equal to $|i + 1|$ are skipped. This is because, at the left boundary of a task, the negative integer i gives width of the task. So, $|i + 1|$ number of cells to the right are occupied by the same task and can be skipped. This gives considerable savings in run time.

Theorems 2 and 3 help in the identification of possible locations of *maximal staircases* and hence reduce the execution time of the algorithm considerably. We do not test each *staircase* for maximal rectangles, as done in [5]. We only test those *staircases* which lie on right boundary of the *area matrix* or the weight of whose origin is more than that of the cell immediately to its right.

As shown in Figure 5, in order to make a *staircase* at location $(x+1, y)$, we need to calculate the number of contiguous empty cells (y" - y) in column $x + 1$. In [5], this number

was calculated by adding one to the height of free cells in previous row at the same column (at location x+1,y+1). So, height information for a full row of data cells need to be kept in memory. In our approach, the positive weight of a cell gives the number of empty locations above that cell directly. As a result we do not need to save height information of the previous row. So, our algorithm needs half the memory as compared to [5].

Extensive placement studies showed that only about 8% of all *staircases* are *maximal staircases*. In Figure 2, out of 58 possible *staircases*, only 4 are *maximal staircases*. These *staircases* are present at points M, N, O and P.

The entire algorithm for finding all maximal empty rectangles in a area specified by a list of rows $\mathcal{R}$ and columns $\mathcal{C}$ is given in listing *FindMaximalRectangles*.

Inputs:
$\mathcal{A}(W \times H)$ Area Matrix. H is height and W is width.
$\mathcal{R}(1:r)$ List of rows of $\mathcal{A}$.
$\mathcal{C}(1:c)$ List of columns of $\mathcal{A}$.
$\mathcal{L}$ List of rows above top horizontal boundary of all placed tasks and bottom row of the area matrix. Also, $\mathcal{L} \subset \mathcal{R}$;
Outputs:
$\mathcal{M}$ List of all maximal empty rectangles.
Begin FindMaximalRectangles($\mathcal{A}, \mathcal{L}, \mathcal{R}, \mathcal{C}$)
$\mathcal{M} = \emptyset$;
for each $\{y | y \in \mathcal{L}\}$ \\ Examine rows in $\mathcal{L}$
$x = 1$;
  while $x \leq |\mathcal{C}|$
  {
    if $W(x,y) < 0$ \\ (x,y) is occupied
      $x = x + |W(x,y)|$; \\ Skip Columns
      skip; \\Skip to end, start new iteration
    end if
    Staircase $\mathcal{S} = MakeStaircase(x,y)$;
    if $((W(x+1,y) < W(x,y))$ OR $(x == C))$
      $\mathcal{M} = \mathcal{M} \cup ExtractMaximalRectangles(\mathcal{S})$;
    end if
    $x++$; \\ Go to next column
  }
end for
return $\mathcal{M}$
end FindMaximalRectangles

Figure 8: Algorithm for Generating MERs

In the listing, *MakeStaircase* is procedure for making a *staircase* at a cell (Section 4.2). Procedure *ExtractMaximal-Rectangles* extracts the maximal rectangles out of a *maximal staircase* (Section 4.3).

5. TIME AND SPACE COMPLEXITY

An FPGA has x number of columns and y number of rows. Let n be the number of rows in the FPGA which lie immediately above top boundary of some tasks. As proved in Theorem 2, the maximum number of rows that need to be scanned to make staircases will be $n+1$. In the worst case, a staircase needs to be constructed at every column in a row. A staircase can be made in constant time. A staircase can be checked for maximal rectangles in a very small amount of time, considered to be a constant [5]. So, time complexity of our algorithm is $\mathcal{O}(xn)$. Upper limit on number of rows that need scanning is y. So, worst case performance of our algorithm is $\mathcal{O}(xy)$.

Number of steps of a staircase is neither more than the number of rows nor more than the number of columns. So, space complexity of our algorithm is $\mathcal{O}(min(x,y))$ [5] which is same as [5]. However, as discussed in Section 4.5, our algorithm uses only half as much memory compared to [5].

In the algorithm given in [5], average case performance is same as the worst case performance. As we will see in the next section, in our version, however, due to Theorems 2 and 3, average run time of the algorithm is much better than its worst case performance.

6. RESULTS

We performed extensive testing to study the behavior of our algorithm. In our experiments, we generated about 10 million staircases after dynamic addition and deletion of tasks. Figure 9 shows the number of skipped rows, 10 shows the number of skipped columns in the scanned rows and 11 shows total number of skipped cells as a function of percentage empty area. Each figure has more than 2000 sample points. Empty area on the FPGA decreases with

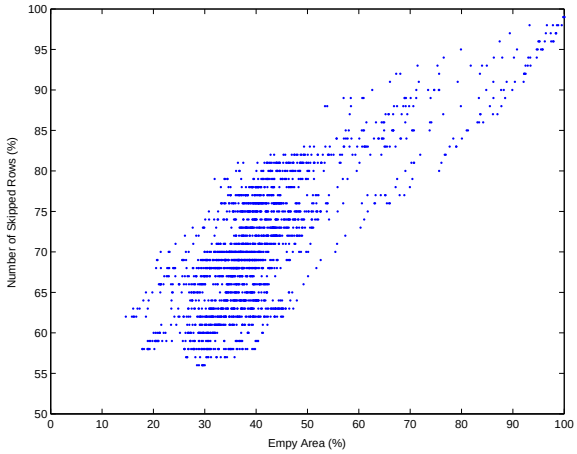

Figure 9: *Number of Skipped Rows Vs Empty Area*

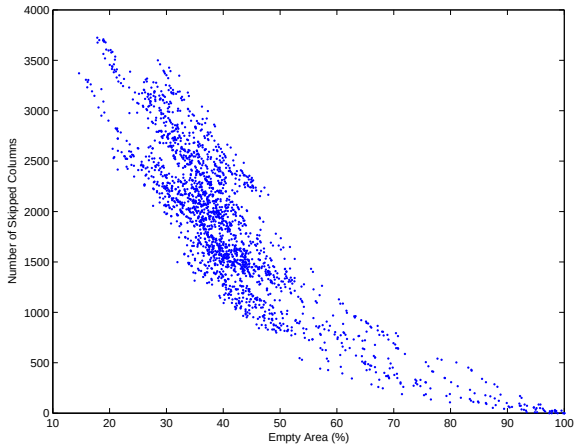

Figure 10: *Number of Columns Vs Empty Area*

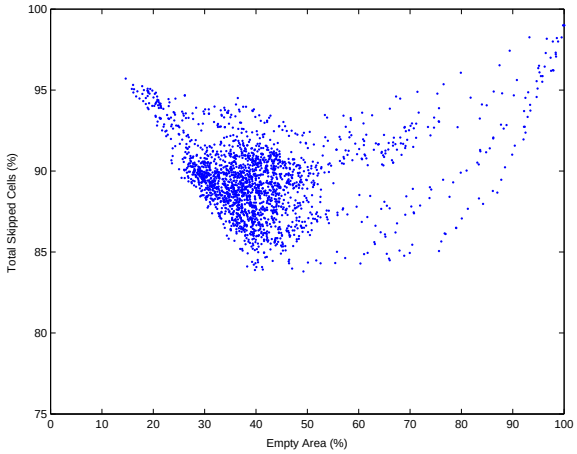

Figure 11: ***Total number of Cells Vs Empty Area***

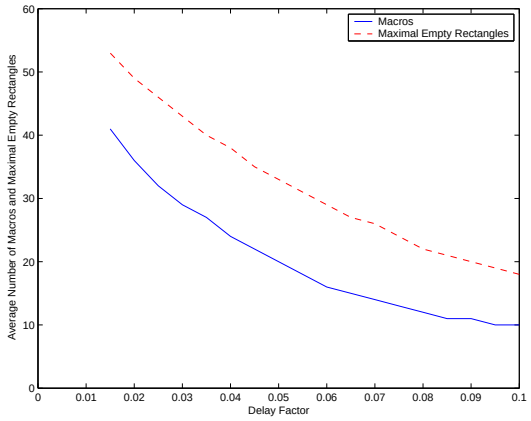

Figure 12: ***Average Number of Tasks and Free Rectangles Vs Delay Factor***

the increase in number of tasks placed on it and vice versa. When there is more empty area on the FPGA, there are less tasks placed on it. As a result, number of skipped rows increases due to Theorem 2 (Figure 9). On the other hand, when there is less empty area on the FPGA, the number of skipped columns increases due to Theorem 3 (Figure 10).

Figure 11 shows the number of total skipped cells as a function of empty area. If the empty area is more on the FPGA, the Theorem 2 becomes dominant and if empty area is less, Theorem 3 dominates. Due to combined effect of both, the number of skipped cells are more than about 85%. So, in all cases, we need to check only 15% of the cells on the FPGA and so our algorithm executes in only 15% time as compared to its worst case performance.

These results shows that our algorithm has much shorter run time as compared to [5] where no row or column can be skipped and all staircases must be tested for finding maximal rectangles in them. This improvement is entirely due to our encoding of the area matrix, theorems 2 and 3 and the heuristics discussed in Section 4.5.

Another interesting relationship is the variation of number of maximal empty rectangles as a function of the tasks placed on the FPGA. Figure 12 shows number of tasks and free rectangles as a function of the delay factor (Delay factor is defined as ratio of maximum time delay between consecutive tasks and the maximum life-time of tasks). Each run performs dynamic placement and removal of 1000 tasks on the FPGA. Height and width of each task is generated randomly between 1 and 25. Life time of each task is also generated randomly between 1 and 1000.

As shown by the Graph 12, the number of maximal empty rectangles is of same order as the number of tasks placed on the FPGA. This implies a significant advantage in managing empty space on the FPGA.

7. CONCLUSION

We presented an efficient algorithm for finding empty area on a FPGA for dynamic placement. We manage free area in the form of maximal empty rectangles. The *staircase* data structure was used for this purpose. We presented an efficient method for finding maximal empty rectangles from *staircases*. In practice, we need to scan only about 15% of the empty cells to make staircases. Our algorithm checks about 8% of the *staircases* for finding maximal rectangles from them. This amounts to tremendous reduction in the run time of dynamic placement algorithms.

8. ACKNOWLEDGMENTS

This work is sponsored in part by the Dayton Area Graduate Studies Institute (DAGSI) and the Air Force Research Laboratory (AFRL) research program under contract number IF-UC-00-07.

9. REFERENCES

[1] M. Gericota, G. Alves, M. Silva, and J. Ferreira. Run-Time Management of Logic Resources on Reconfigurable Systems. In *Proc. of Design, Automation and Test in Europe*, Mar. 2003.

[2] M. Handa and R. Vemuri. Area Fragmentation in Reconfigurable Operating Systems. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, Jun. 2004.

[3] P. Healy and M. Creavin. An Optimal Algorithm for Rectangle Placement. In *Technical Report UL-CSIS-97-1. Dept. of Computer Science and Information Systems, University of Limerick, Limerick, Ireland*, 1997.

[4] M. Orlowski. A New Algorithm for the Largest Empty Rectangle. In *Algorithmica*, volume 5, pages 65–73, 1990.

[5] J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for Empty Spaces in Large Data Sets. In *Theoretical Computer Science*, volume 296(3), pages 435–452. Elsevier Science Publishers Ltd., 2003.

[6] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of computers- Special Issue on Reconfigurable Computing*, volume 17(1), pages 68–83, Jan.-Mar. 2000.

[7] H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing . In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr. 2003.

[8] A. Aggarwal and S. Suri. Fast Algorithms for Computing the Largest Empty Rectangle. In *Proceedings of the third annual symposium on Computational geometry*, pages 278–290, Jun. 1987.