An efficient algorithm for free resources management on the FPGA

Yi Lu, Thomas Marconi, Georgi Gaydadjiev and Koen Bertels
Computer Engineering Lab., TU Delft, The Netherlands
$\{yilu, thomas, georgi\}$@ce.et.tudelft.nl, k.l.m.bertels@tudelft.nl

Abstract

Finding the available empty space for arrival tasks on FPGAs with runtime partially reconfigurable abilities is the most time consuming phase in on-line placement algorithms. Naturally, this phase has the highest impact on the overall system performance. In this paper, we present a new algorithm which is used to find the complete set of maximum free rectangles on the FPGA at runtime. During scanning, our algorithm relies on dynamic information about the edges of all already placed tasks. Simulation results show that our algorithm has 1.5x to 5x speedup compared to state of the art algorithms aiming at maximum free rectangles. In addition, our proposal requires at least 4.4x less scanning load.

1 Introduction

In the majority of current FPGA-based reconfigurable systems the FPGAs are used as slave components that offload the main general purpose processor. In such scenarios the FPGAs execute a single configuration at any given moment. When required, the application reconfigures the complete FPGA. The reconfigurability makes these systems easily adaptive to different applications domains. The reconfiguration process of the complete FPGA, however, introduces long reconfiguration times and increased power consumption. In recent years, with the development of the partially reconfigurable FPGAs, this problem can be addressed by using the partial reconfiguration support to reconfigure only the necessary part of the FPGA when required. In such partially reconfigurable systems, an hardware task is loaded into (or removed from) the FPGA individually without interfering with any other tasks running on the same FPGA. In many cases, such systems have runtime constraints and the sequence of the hardware tasks is unknown in advance. In all on-line task placement algorithms designed for such systems, determining and maintaining the free space on the FPGA is the most time-consuming process. This fact validates the high demand of efficient algorithms to manage the free FPGA space.

Because most of the hardware tasks can be fitted in a rectangular shape, the free FPGA space is usually recorded as a set of rectangles. There are two types of rectangles: the non-overlapping rectangles and the maximum rectangles. In general, it is more time-consuming to maintain a set of maximum rectangles than a set of non-overlapping rectangles. Maintaining a set of maximum rectangles, however, increases the possibility to fit an arrival task on the FPGA [2]. Every time a task is loaded or removed, the information about the free FPGA space has to be updated. During this update process all current maximum rectangles will be recreated. In addition, the fragmentation and defragmentation problems of the free space will be solved, which guarantees the on-line placement efficiency and correctness.

In this paper, we propose a novel algorithm to find the complete set of maximum free rectangles at runtime. The main contributions of this paper are:

- a new mechanism to find a complete set of maximum free rectangles on the FPGA;
- better performance, supported by simulation results, compared to other state of the art approaches.

In section 2, related work is presented. Then, we detail our algorithm in section 3. In section 4, we present the simulation results and evaluate performance of our and two previously proposed algorithms. Finally, we conclude this paper and discuss future directions in section 5.

2 Related Work

In 1999, Bazargan et al. [2] proposed their on-line task placement approach. This approach stores the free space of the FPGA as a set of non-overlapping rectangles and can achieve high speed but at the cost of low placement quality. Walder et al. [8] improved Bazargan's on-line algorithm by delaying the decision about split heuristic until a new task arrives. In addition, a hash matrix was proposed to store the non-overlapping rectangles to guarantee a constant search time. In [5], Handa et al. proposed an algorithm to find empty space on the FPGA. In their algorithm, the FPGA surface is modeled as a 2D array of configurable units, referred as "area matrix". Their algorithm starts with encoding the matrix. Thereafter, all maximum staircases [4] are found based on the encoded information. Finally the maximum free rectangles are extracted from each maximum staircase. In [3], Cui et al. used the same 2D FPGA surface model but with different encoding information. The authors defined MKE points to utilize the scanning process while looking for the maximum free rectangles. In [7], Tomono et al. proposed an online placement approach, which takes the module connectivity to the reminder of the system into account. In their approach, the staircase algorithm [5] is reused to find the complete set of maximum free rectangles. Tabero et al. [6] used vertex-lists to store free space, where each vertex is a possible location for an input task. Each arriving task is placed by selecting a vertex corresponding to appropriate size from the list. In [1], Ahmadinia et al. proposed a new way to manage the free FPGA resources, that only stored the information about used space.

In all algorithms above two basic approaches to manage free space with rectangular shape can be defined: the tracing approach and the scanning approach. In the tracing approach, only non-overlapping rectangles can be created and used. Because there is no overlap between any two rectangles, all geometry operations are limited to the current rectangle only. A tree data structure is used to store the rectangles. Algorithms using this approach, e.g. [2, 8, 1], achieve shorter algorithm execution time, but low overall placement quality (based on the task rejection rate). Non-overlapping rectangle based approaches can be unable to fit a new arrival task although there is enough space available as shown in the simple example depicted in figure 1(a). However, when using the scanning approach, whose output is the complete set of maximum free rectangles, the arrival task will be placed as shown in figure 1(b).

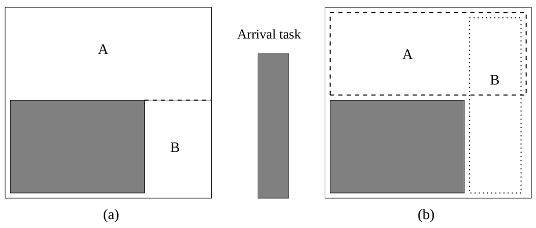


Figure 1. Allocation of an arrival task

In the reminder of this paper, only the scanning approach will be considered due to its higher placement quality. By using the available information, e.g. the encoding information of the FPGA surface model [5, 3], the scanning approach can perform runtime scanning. According to [5], the number of overlapping maximum rectangles and non-overlapping rectangles are nearly the same after a significant number of experiments. So the memory requirements for storing them and time for searching in the list of rectangles will be similar. Meanwhile maintaining maximum rectangle achieves 8% placement quality increase [2].

In this paper, we propose a novel algorithm using the scanning approach to find the complete set of maximum free rectangles. The details about the algorithm are presented in the next section.

3 Flow scan algorithm

The algorithm proposed in this paper is called *Flow Scan* (FS). The FS algorithm is characterized by fast FPGA free space management. In this section we first present some definitions needed for the discussion to follow. Next, the data structures (linked lists) that store runtime information about the free space is introduced. Thereafter, the flow scan processing and the operations on the linked lists are described. Last the proof of completeness is presented.

3.1 Definitions

In-edge and out-edge: For each placed task, its lower Y coordinate is defined as *in-edge*. The *out-edge* corresponds to the higher Y coordinate of the same task. The bottom and top of the FPGA area are defined as out-edge and in-edge by default respectively. The scanning flow direction is from in-edge to out-edge, as shown in the figure 2.
Rectangular well (RW): During the scanning process, some temporally rectangles without top lines are created, we define such rectangles as *rectangular wells*.
Formed rectangular well (FRW): Any RW that can only be expanded upwards are defined as FRW as shown in the figure 2. If there are several RWs with the same X coordinates created during the scanning process, only the FRW is recorded. An example will be discussed in section 3.3.
Maximum free rectangle: It is defined as a rectangle whose top, bottom, left and right edge can not be expanded. It is abbreviated as (left, right, bottom, top) in this paper, e.g. (0, 100, 0, 20) for the maximum free rectangle available at the bottom of figure 2.

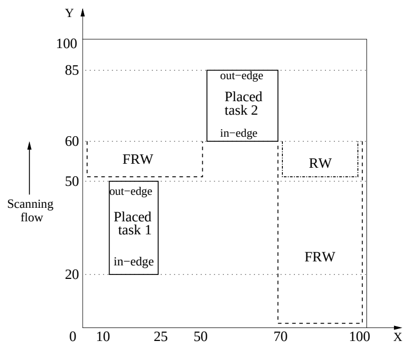
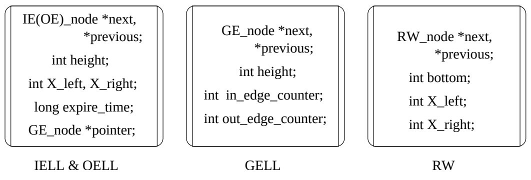


Figure 2. FGPA model for the FS

3.2 Data structure

In our algorithm we use linked lists to store the required information. We defined 4 different linked lists: general edge linked list (GELL), in-edge and out-edge linked lists (IELL and OELL), and rectangular well linked list (RWLL). The node data structures for those linked lists are shown in figure 3. A GELL node consists of the edge height at which one or more edges are present. In addition two edge counters are present to store the number of in-edges and out-edges on that height. A node of IELL or OELL consists of the height and the X coordinates of the edge, the expire time of the corresponding task, and a pointer to the

GELL node which represents the same height. This pointer is used to updating the corresponding edge counter when a new edge is inserted or existing one is removed. RWLL stores all current FRWs. A RW node in the RWLL stores the lower Y and both X coordinates of the FRW. In figure 4, the linked lists representing the situation as depicted in figure 2 is shown.



```
IE(OE)_node *next,
        *previous;
  int height;
  int X_left, X_right;
  long expire_time;
  GE_node *pointer;
```

```
GE_node *next,
       *previous;
  int height;
  int in_edge_counter;
  int out_edge_counter;
```

```
RW_node *next,
       *previous;
  int bottom;
  int X_left;
  int X_right;
```

Figure 3. Data structure

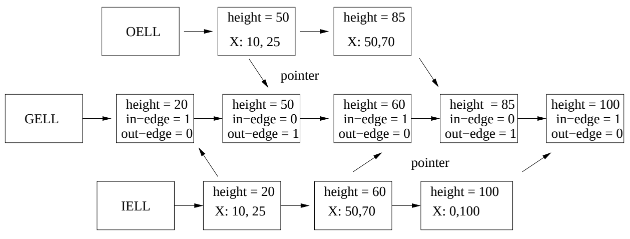

OELL
height = 50 X: 10, 25
height = 85 X: 50,70
pointer
GELL
height = 20 in−edge = 1 out−edge = 0
height = 50 in−edge = 0 out−edge = 1
height = 60 in−edge = 1 out−edge = 0
height = 85 in−edge = 0 out−edge = 1
height = 100 in−edge = 1 out−edge = 0
pointer
IELL
height = 20 X: 10, 25
height = 60 X: 50,70
height = 100 X: 0,100

Figure 4. Linked lists

3.3 Flow scan processing

There are two basic scan procedures in the FS algorithm, the in-edge processing and out-edge processing. The in-edge processing happens when the scanning flow reaches an in-edge and the out-edge processing is called when leaving an out-edge. In the in-edge processing, if a FRW is overlapped with an in-edge in the X direction, a maximum free rectangle is created by adding to the FRW a top line at the height of the in-edge. Only when the scanning process reaches an in-edge, the search for overlapped FRWs will start and if any found the maximum free rectangle will be created. In the cases $FRW_L <$ in-edge$_L < FRW_R$ or $FRW_L <$ in-edge$_R < FRW_R$[1], at most two new RWs can be created for the non-overlapping area within the FRW. If the width span (the length along the X axis) of a FRW is fully covered by an in-edge, no FRW will be generated. In the out-edge processing, only one new FRW is created. Its bottom has the same height as the out-edge.

A simple example shown in figure 5 is used to clarify the process in the following. In the beginning, an initial FRW is created at the bottom of the 2D FPGA area. The bottom of this FRW is 0 and it covers the whole width of the FPGA area, as shown in figure 5(a). The scan process will reach the in-edge of task 1 at height of 20 in the Y direction (shorthand **At height = 20**:), the initial FRW is overlapped with this edge in X direction, so it becomes a maximum free rectangle (0, 100, 0, 20). Thereafter, two new RWs are created for the non-overlapping area as explained above. Because both of them can only be expanded upwards, they are FRWs, as shown in figure 5(b), the FRW_1 and FRW_2. This step is completed by recording the two FRWs into RWLL and outputting the one maximum free rectangle found: (0, 100, 0, 20).

At height = 50: the out-edge of task 1 is met at this level, so the out-edge processing is performed, which creates a new FRW: FRW_3 shown in figure5(c).

At height = 60: the in-edge processing is initiated. Because the FRW_2 and FRW_3 are overlapped with the in-edge of task 2, two maximum free rectangles, (25, 100, 0, 60) and (0, 100, 50, 60), are found and generated. The FRW_4, FRW_5 and FRW_6 are created for the non-overlapping areas. As shown in figure5(d), there is another temporal RW_{temp}, which is created for the non-overlapped area between FRW_3 and the in-edge. As mentioned earlier, among the existing RWs with same X coordinates, only covering FRW is recorded, which is FRW_6 in our example.

At height = 85 and height = 100 [2]: the FRW_7 is created at the out-edge of task 2. When reaching the top edge (100) all existing FRWs are transferred to maximum free rectangles with top at Y = 100.

During the scanning process described above, totally eight maximum rectangles were found: (0,100,0,20), (0,100,50,60), (0,10,0,100), (0,50, 50, 100), (0,100,85,100), (25,100,0,60), (25,50,0,100) and (70,100, 0,100). The overall algorithm is shown in the figure 6.

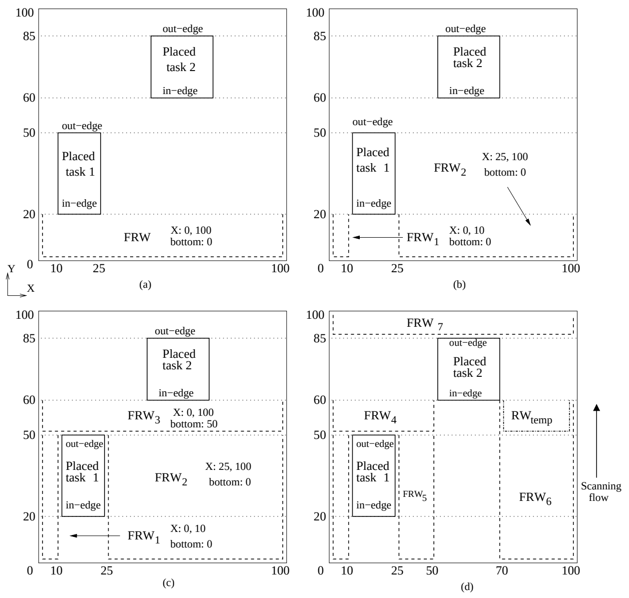


Figure 5. Scan processing

[1]The FRW_L represents the left side of the FRW and FRW_R is the right side; similar considerations hold for the in-edge.

[2]Please note that height=100 is the end of the FPGA area, in our example, which is handled as an in-edge.

FS Algorithm

```
/* input parameters: IELL, OELL, GELL, RWLL
/* output: a list of complete maximum rectangles
IELL_current_position = IELL;
OELL_cuttent_postition = OELL;
while(GELL_temp = GELL != NULL)
    if(GELL_temp−>in_edge_counter >= 1 &&
       GELL_temp−>out_edge_counter >= 1)
          out_edge_processing(OELL_current_position, GELL_temp−>height);
          in_edge_processing(IELL_current_position, GELL_temp−>height);
    else if(GELL_temp−>in_edge_counter>= 1)
          out_edge_processing(OELL_current_position, GELL_temp−>height);
    else if(GELL_temp−>out_edge_counter>= 1)
          in_edge_processing(IELL_current_position, GELL_temp−>height);
    GELL_temp = GELL_temp −> next;
    end if;
end while;
end FS;
out_edge_processing(OELL_ref, height)
    while(OELL_ref−>height == height)
        create_FRW(bottom, left, right);
        OELL_ref = OELL_ref−>next;
    end while;
end processing;
in_edge_processing(IELL_ref, height)
    while(IELL−ref−>height == height)
        if(IELL_ref−>X_left(X_right) is overlapped with nodes in RWLL)
            record_maximum_rectangle(overlapping_RW_node, IELL_ref−>height);
            create_FRW(bottom, left, right);
        end if;
        IELL_ref = IELL_ref−>next;
    end while;
end processing;
```

Figure 6. FS Algorithm

3.4 Operation on linked lists

There are two types of linked list operations used in our algorithm. More precisely, the linked list update and linked list search. The search operation checks all recorded edge nodes and finds all maximum free rectangles existing on the FPGA. The algorithm starts to search the GELL. When checking a node in the GELL, the in-edge and out-edge counters show the number of in- and out-edges on the current height level. A simple example can be found in the figure 4 where the first item represents the situation at height = 20 with only one in-edge as represented by the two counter values. Thereafter, the algorithm searches the OELL or (and) IELL according to the values of the counters. When searching OELL and IELL, the FRWs are created and the maximum free rectangles are found as described in section 3.3.

The linked list update operation adds (deletes) edge nodes into (from) the lists and adjusting the edge counters to right value. All edge nodes are ordered in increased height order. When a new task arrives, two edge nodes standing for its in- and out-edges are created and added into IELL and OELL separately. Next, if the GELL already has a node characterized by the same height as that of any of the edges, the corresponding counter is incremented by 1. Otherwise, a new node reflecting that height is added at the right position. When a task completes its computation, the related two edge nodes in OELL and IELL are removed while the edge counters in related GELL nodes are adjusted using the pointers in the OELL and IELL nodes. If in a GELL node both edge counters equal ’0’, this node will be removed from the list. Otherwise, there are still other edges on this height.

3.5 Proof of completeness

In this section, we prove two theorems which guarantee that the FS algorithm finds the complete set of true maximum rectangles.

Theorem 1: *Generated FRWs always start at an out-edge height and the set of FRWs created on any edge* **i** *is complete and correct.*

Assuming the bottom of a FRW is not positioned on the height of any out-edge implies that this FRW can still be expanded downwards until it reaches an out-edge. This is contradictory to the definition of the FRW (a FRW can only be expanded upwards). This proves that all FRWs start from an out-edge.

Assume there is a missing or incorrect FRW created when our algorithm scans edge i. (A) in case of an out-edge (as shown in figure 7(a)) the L and R are the left side and right side of the free space at out-edge i. In the out-edge processing, one FRW is created with X dimensions equal to L and R. If there is a missing or incorrect FRW, e.g. FRW_m, one of the conditions (L$<$ L’ and R’$\leq$ R) or (L$\leq$ L’ and R’$<$ R), should hold. This means the free space span ([L’..R’]) is smaller than the real free space available ([L..R]). This proves the assumption of missing FRW at out-edge i wrong, hence the single FRW created from an out-edge is correct. (B) For any in-edge: Assume a FRW is missing, e.g. the FRW_i in figure 7(b), or its width span does not cover the full non-overlapping area. The bottom of FRW_i is on the out-edge of Task$_j$. As described in in-edge processing, the RWs are created for the non-overlapping area from FWRs (FRW_j in our example) overlapped with the in-edge (in-edge$_i$). If FRW_i is missing implies that the FRW_j created from the out-edge j is incorrect. More precisely, it does not contain the area that FRW_i occupies (or a part of that area when FRW_i is not correctly created). This contradicts with the proof about the FRW from an out-edge generation presented above. This proves the assumption above wrong. □

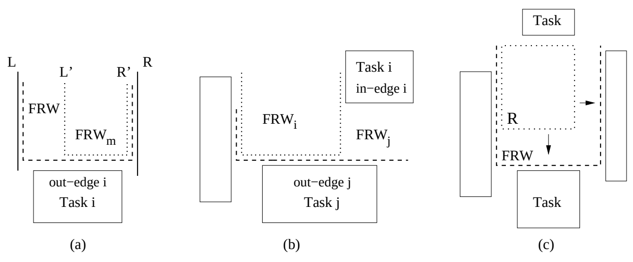


Figure 7. Contradiction situation

Theorem 2: *There is a one-to-one relationship between the set of FRWs and the complete set of maximum free rectangles.*
Assume two FRWs have overlapping area, e.g. FRW and FRW_m as shown in figure 7 (a). This means that one of them can be expanded horizontally (FRW_m in this example), contradicting with the FRW definition. So, any two FRWs can not overlap, proving each FRW as unique.

Assume a maximum free rectangle R is not from any FRW. So this rectangle can still be expanded, as shown in figure 7(c). This is a contradiction with the definition of the maximum free rectangle as presented earlier. This proves that any maximum free rectangle is generated from a FRW.

Assume there is a FRW that does not become a maximum free rectangle after the scanning process. This means there is no in-edge overlapped area above this FRW. This contradicts with the fact that the top border of the FPGA area is defined as the highest in-edge with width span equal to the FPGA area width. This implies that all FRW will become maximum free rectangles after the scanning flow is completed. $\square$

Overall, the second theorem describes the one-to-one relationship between FRWs and maximum free rectangles. So if the set of FRWs is complete and correct, the whole set of maximum free rectangles is found completely and correctly. Thanks to the first theorem which guarantees that the complete and correct set of FRWs is created. So, the FS algorithm finds the whole set of maximum free rectangles completely and correctly.

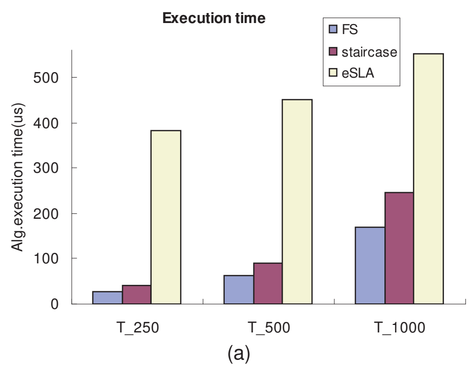
4 Experimental evaluation

We performed simulations in order to evaluate performance of our FS algorithm, the staircase algorithm [5] and the enhanced SLA (eSLA) algorithm [3]. The three algorithms were implemented in C, and evaluated under Linux 2.6 running on Intel Pentium(R) 4 CPU 3.00GHz with 2GB main memory.

In each simulation run, 10000 tasks were generated randomly since our primary concern was not the rejection rate evaluation but fair execution time comparison. We integrated the three scanning algorithms in the same simple online placement algorithm. This placement algorithm uses first fit policy to find a suitable allocation for the arrival tasks from the set of maximum free rectangles generated by the scanning algorithm. Before each simulation, a tracing process using the placement algorithm equipped with one of the scanning approaches is performed. We saved the tracing output which contains the partitioning information during placement in a trace to ensure the three scanning algorithms used exactly the same partitioning during execution. This trace was generated as follows: we run the placement algorithm 10000 times and store a single task into the trace at each run. We start from the complete size of the 2D FPGA area (100x100 configurable units (CUs)) and we create each new task by using the output of the previous scanning algorithm execution, which corresponds to the current complete set of maximum free rectangles. One of the maximum free rectangles is selected randomly. The size of the new task is randomly generated within the selected maximum free rectangle. Considering the arrival time each task is assigned a random number between $[5..25]$ time units. In respect to the task life time 3 ranges were used: T_{250}, T_{500}, and T_{1000}. For T_{250} the task life time is randomly chosen from the time interval $[5..250]$, for T_{500} the $[251..500]$ is used, and for T_{1000} $[501..1000]$. All the above information about the partitioning and the new tasks is saved in our trace. We used the generated trace to evaluate the scan time of the three algorithms. Please note that the selection of which algorithm is used to generate the trace above is irrelevant for our study because all maximum free rectangles are queued in the same order and the first fit is used.

4.1 Execution time

The algorithm executed every time when a task arrives or one is removed. In our simulation with 10000 tasks, the algorithms are executed approximately 15000 times. In the figure 8, the average execution time of a single algorithm call and its execution time distribution are presented. As shown in the figure 8 (a), our algorithm has shortest execution time compared to the other two algorithms for all three task sets. The eSLA has the longest execution time in all simulations. The reason is that in both eSLA and staircase algorithms, in order to find all maximum free rectangles, the information stored in bigger number of CUs should be accessed. In addition, during the update process they have to adjust the information in all related CUs. In our algorithm, only the task edges are processed, which is a relatively smaller number. The worst case for our algorithm is when all edges of *n* placed tasks are located on different heights, implying $2n$ nodes have to be accessed. This makes the worst case complexity of our algorithm $O(2n)$. On average our algorithm is 1.5x faster than staircase and 5x faster than eSLA respectively. In figure 8(b), the distributions for single run time of algorithms are given. For example, in figure 8(b), the highest point of the curve representing the FS algorithm, indicates that 50% of the algorithm calls (around 7500) complete in the time interval between 20μs and 40μs. For short task lifetimes (figure 8(b)) FS has execution times clearly concentrated on the left side of the graph. This is due to the fact that with short task life times a low number of tasks is present on the FPGA and the total number of edges to be processed by the FS algorithm is small. For all three ranges of life times, the density of the FS algorithm samples is higher in the shorter time periods compared to the other two algorithms, similar as the situation shown in figure 8(b).



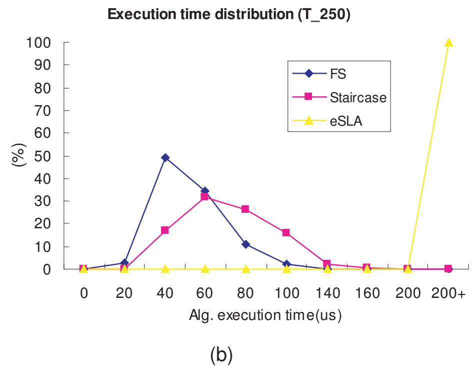


Figure 8. Time issue

4.2 Scanning load

In the staircase and the eSLA algorithms, all CUs are encoded. The algorithms use the encoded information to find the maximum rectangles. In our algorithm, we use linked lists to record information and to find maximum free rectangles. The scanning load is defined as the number of CUs (or nodes of linked list) have to be accessed during algorithm execution. As shown in the table 1, for each update, the staircase modifies at least 191 CUs and the eSLA minimum 160 CUs. In our algorithm in the worst case only 3 nodes will be added (deleted) into (from) the GELL, OELL and IELL respectively. During the scanning process (looking for the complete set of maximum free rectangles) the number of nodes need to be checked by our FS algorithm is much lower than the number of CUs need to be visited in the other two algorithms, as shown in the last collum of the table 1. The large number of CUs for the eSLA algorithm is many CUs are checked several times for different rectangles in the same scan iteration.

5 Conclusion and future work

In this paper, we proposed a new algorithm for finding the complete set of maximum free rectangles during online FPGA placement. Our experimental results have shown that the Flow Scan algorithm has better performance compared to state of the art algorithms providing the same functionality. In the future, our work will focus on: (i) integrating the Flow Scan algorithm in previously proposed on-line placement technique; (ii) apply the FS algorithm in task scheduling schemes.

		No. of update CUs(nodes)	No. of scan CUs(nodes)
T_{250}	FS	≤ 3	38
	staircase	502	463
	eSLA	463	20379
T_{500}	FS	≤ 3	70
	staircase	321	597
	eSLA	255	21547
T_{1000}	FS	≤ 3	130
	staircase	191	573
	eSLA	160	20852

Table 1. Scanning load

Acknowledgment: This work is sponsored by the hArtes project (IST-035143) supported by the Sixth Framework Programme of the European Community under the thematic area "Embedded Systems".

References

[1] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. van der Veen. Optimal free-space management and routing-conscious dynamic placement for reconfigurable devices. In *IEEE Transactions on Computers*, volume 56, pages 673–680, May 2007.

[2] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. In *In IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.

[3] J. Cui, Q. Deng, X. He, and Z. Gu. An efficient algorithm for online management of 2d area of partially reconfigurable fpgas. In *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*, pages 129–134, Nice, France, April 2007.

[4] J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for empty space in large data sets. In *In Theoretical Computer Science*, volume 296(3), pages 435–452. Elsevier Science Publishers Ltd., 2003.

[5] M. Handa and R. Vemuri. An efficient algorithm for finding empty space for online fpga placemen. In *In Proceedings of the 41st annual conference on desing automation*, pages 960–965, San Diego, June 2004.

[6] J. Tabero, J. Septien, H. Mecha, and D. Mozos. Low fragmentation heuristic for task placement in 2d rtr hw management. In *14th International Conference on Field Programmable Logic and Application, FPL 2004*, pages 241–250, Antwerp, Belgium, Sept. 2004.

[7] M. Tomono, M. Nakanishi, S. Yamashita, N. Nakajima, and K. Watanabe. A new approach to online fpga placement. In *In 40th annual conference on Information Sciences and Systems*, pages 145–150, Princeton, USA, March 22-24 2006.

[8] H. Walder, C. Steiger, and M. Platzner. Fast online task placement on fpgas: Free space partitioning and 2d-hashing. In *In Reconfigurable Architectures Workshop (RAW)*, Nice, France, April 2003.