A Novel Fast Online Placement Algorithm on 2D Partially Reconfigurable Devices

Thomas Marconi, Yi Lu, Koen Bertels, Georgi Gaydadjiev

Computer Engineering Lab, Delft University of Technology
{thomas,yilu,koen,georgi}@ce.et.tudelft.nl

Abstract—**In this paper, we propose a new strategy for online placement algorithm on 2D partially reconfigurable devices, termed the Quad-Corner(QC). The main differences between our algorithm and related art are quad-corner spreading capability and dynamical searching sequences. Moreover, existing algorithms do not evaluate their algorithms with real hardware tasks; we do experimentations with real hardware tasks on a real FPGA. Our proposal achieves better placement quality and fast online placement compared to existing approaches. Experiments with real workloads (e.g. MDCT, matrix multiplication, hamming code, sorting, FIR, ADPCM, etc) on Virtex-4 show that the QC not only has 78 % less penalty and 93 % less wasted area than the existing algorithms on average but also has lower runtime overhead.**

I. Introduction

Current reconfigurable devices have the ability to reconfigure parts of their hardware resources without interrupting normal operation of the remaining fabric. Researchers are challenged to exploit the ability of these devices to reduce power consumption, reduce cost, increase performance, and many more. However to get benefits of partial reconfiguration is not for free, many problems have to be solved, such as: reconfiguration time overhead reduction, hardware task placement and scheduling, inter-tasks communication, tasks-IOs communication, etc.

The placement algorithms need to find locations for placing arrival tasks and to maximize the utilization of the resources. Because the running time of online algorithms is considered as an overhead for overall execution time of the applications, therefore not only placement quality but also the speed of the algorithm should be addressed. As optimal algorithms are too slow to be used in practice, so the heuristic strategies are needed. In one hand, high placement quality algorithms are slow. In the other hand, fast algorithms have poor placement quality. Hence discovering a high quality, fast placement strategy is challenging. To address this challenge, simple yet effective algorithms are required.

Many authors have already proposed algorithms to deal with placement issues. Some of them focus on efficient resource utilization, while others are interested in fast algorithms. In this paper, we focus on lowering runtime overhead. In [1], Bazargan et al. presented a fast online placement algorithm. The algorithm uses a heuristic to decide if a free rectangle is split vertically or horizontally upon task placement. Regardless of the performance of the heuristic, the possibility of conducting wrong split exists, resulting on diminished quality of future placements. In [2], three techniques for speeding up online placement algorithms: *Merging Only if Needed* (MON), *Partial Merging* (PM), and *Direct Combine* (DC) were proposed. Furthermore, *Intelligent Merging* (IM) algorithm was proposed by applying these techniques. Instead of merging all fragmented free space blocks, the algorithm only merges on demand to decrease the runtime overhead. The algorithms that use splitting and merging in managing free area need to merge free area for accommodating arrival tasks.

Moreover, existing algorithms are not evaluated with real hardware tasks. In this paper, we do experimentations with real hardware tasks on a real FPGA. The main differences between our proposed algorithm and other algorithms are quad-corner spreading capability and dynamical searching sequences. Our algorithm tries to spread hardware tasks to the four corners of the devices to leave as large as possible free area in the middle for better placement of future tasks.

The main contributions of this paper are:

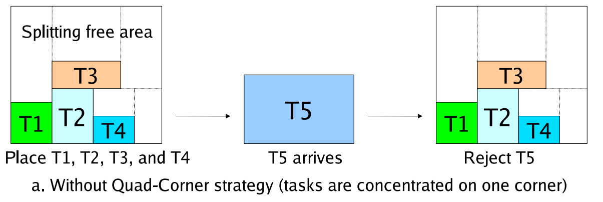
- a novel strategy for online placement algorithm;
- lower runtime overhead and better placement quality compared to other related approaches.

The rest of the paper is organized as follows. The details of our QC strategy are presented in Section II. In Section III, we present the evaluation of the proposed algorithm. Finally, in Section IV, we summarize the paper.

II. The QC strategy

A. Basic idea of quad-corner strategy

The existing strategies tend to place arrival tasks concentrating on one corner and (or) split free area into many small segments as shown in Figure 1a. These can lead to the undesirable situation that a task cannot be allocated even if there would be sufficient free area available. Because of these problems, task T5 can not be accommodated as shown in this motivating example in Figure 1a. As a consequence, the reconfigurable device is not well utilized (waste of resources). Furthermore, task T5 has to wait for execution or to be executed by the host processor due to this inefficient placement, hence the application will be slowed down (performance degradation). To address these problems, we spread hardware tasks close to the four corners of the devices as illustrated in Figure 1b. There are two main advantages of this strategy as shown in Figure 1b: (1) it reserves a lot of free area in the middle of the device; (2) it solves splitting free area problem. As a result, both the reconfigurable device utilization and the system performance will be increased.


Splitting free area
T1
T2
T3
T4
T5
Place T1, T2, T3, and T4
T5 arrives
Reject T5
a. Without Quad-Corner strategy (tasks are concentrated on one corner)
Place T1, T2, T3, and T4
T5 arrives
Place T5
b. With Quad-Corner strategy (tasks are spread on four corners)

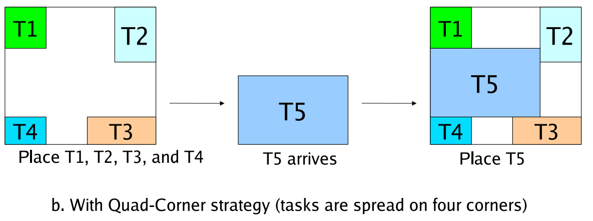


Fig. 1. Basic idea of quad-corner strategy

B. Two-dimensional reconfigurable device and task models

A two-dimensional reconfigurable device, denoted as *RD(H,W)*, consists of *HxW* homogeneous reconfigurable units arranged in a two-dimensional array of height *H* and width *W* and an interconnect between the units. A reconfigurable unit in row *r* and column *c* is represented by *ru(r,c)*, for $0 \leq r \leq H\text{-}1$ and $0 \leq c \leq W\text{-}1$, with *ru(0,0)* as the upper left corner.

Our task model has rectangle shape and includes all the required reconfigurable units and interconnect resources. We assume tasks are independent, with no precedence and direct connection among them.

C. Task types

To make the free area in the middle of the device as large as possible, we force our algorithm to spread hardware tasks close to the four corners of the devices. To support this idea, therefore we define four ways of placing tasks: starting from upper left corner, upper right corner, lower right corner, and lower left corner. We divide tasks into four different task types (Figure 2): upper left task (ULT), upper right task (URT), lower right task (LRT), and lower left task (LLT). A *THxTW* task in a two-dimensional reconfigurable device *RD(H,W)* is a group of reconfigurable units belonging to *RD(H,W)*, with task height *TH* and task width *TW*, such that $1 \leq TH \leq H$ and $1 \leq TW \leq W$. The task has an origin reconfigurable unit *ORU=ru(OR,OC)* and two alternative placement positions for accommodating future tasks. The two alternative positions are a *horizontal alternative position* (HAP) and a *vertical alternative position* (VAP) as origin reconfigurable units for future tasks. OR and OC denote origin row and origin column respectively.

D. Initial placement positions

To spread tasks to the corners for creating as large as possible free area in the middle, we propose four initial placement positions for accommodating tasks on an empty *WxH* two-dimensional reconfigurable device: upper left task initial *ULTI=ru(0,0)* (for upper left tasks), upper right task initial *URTI=ru(0,W-1)* (for upper right tasks), lower right task initial *LRTI=ru(H-1,W-1)* (for lower right tasks), and lower left task initial *LLTI=ru(H-1,0)* (for lower left tasks).

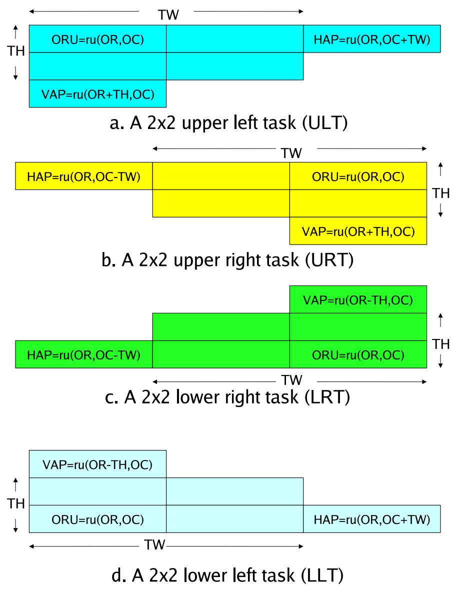


Fig. 2. Examples of four task types and their alternative placement positions

E. Data structures

We use a 2D matrix *RD(r,c)* (RD matrix) to represent the FPGA area, defined as: *RD(r,c)=0* if *RD(r,c)* is not occupied (free) or *RD(r,c)=1* if *RD(r,c)* is occupied (used), where $0 \leq r \leq H\text{-}1$, $0 \leq c \leq W\text{-}1$, and *RD(0,0)* is the element of upper left corner.

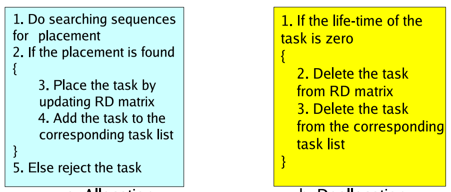
During placement, the software implementation of the proposed algorithm maintains four lists: a upper left task list (for storing upper left tasks), a upper right task list (for storing upper right tasks), a lower right task list (for storing lower right tasks), and a lower left task list (for storing lower left tasks).

F. Searching sequences for placement

To accommodate tasks, the algorithm needs to search four task lists as mentioned above. In order to pack tasks more compactly, the algorithm searches placements in all different task lists based on the sizes of arrival tasks. There are four different searching sequences for placement: upper left corner first (for very large tasks), upper right corner first (for large tasks), lower right corner first (for medium size tasks), and lower left corner first (for small tasks). The strategy tries to group tasks based on their sizes. This way, the algorithm picks the corner which contains tasks that are similar in size as the task that needs to be placed. For example, finding placements for very large tasks using upper left corner first sequence are fastest. The reason for this is that the algorithm finds a placement starting from the location where mostly contain very large tasks are mapped. This strategy can also increase the placement quality since we group tasks based on their sizes for better compacting purposes. In this paper, we consider serial implementation of the list search. Since the four task lists can work independently, searching task lists can be executed in parallel in a future implementation.

G. The algorithm

The pseudocode of the proposed Quad-Corner algorithm for allocation is shown in Figure 3a. In line 1, the algorithm searches dynamically different possible placements according to its size until it finds the appropriate placement. This strategy reduces the algorithm execution time and at the same time increases its placement quality by finding placements in the specific area and placing tasks as close as to the specific group. If the algorithm finds the placement position, it places the task starting from this position by updating the RD matrix (line 3) and adds the task to its corresponding task list (line 4). If the algorithm cannot find the placement, it rejects the task (line 5).



```
1. Do searching sequences
for  placement
2. If the placement is found
{
    3. Place the task by
    updating RD matrix
    4. Add the task to the
    corresponding task list
}
5. Else reject the task
```

a. Allocation

```
1. If the life-time of the
task is zero
{
    2. Delete the task
    from RD matrix
    3. Delete the task
    from the corresponding
    task list
}
```

b. Deallocation

Fig. 3. The pseudocode of QC strategy

The pseudocode of the proposed Quad-Corner algorithm for deallocation is shown in Figure 3b. In line 1, the algorithm checks the life-time of the task. If the life-time is zero (finished tasks), the algorithm deletes the task from RD matrix (line 2) and its corresponding task list (line 3).

III. Evaluation

To evaluate the proposed algorithm, a discrete-time simulation framework was constructed in C. The framework was compiled and run under Linux on a Pentium-IV 3.4 GHz PC. Since the algorithms are online, the information about new tasks is unknown until their arrival time. We assume that each task should be placed at its arrival time and is rejected when it could not be placed. If a task is rejected, the equivalent function should be executed in software by the host processor and hence a penalty is incurred. We use task set $REJECT$ to represent tasks which are rejected from all task set TS. The volume of a task t_i that has a width w_i reconfigurable units, height h_i reconfigurable units and life-time lt_i time units is defined as $v_i(t_i) = w_i.h_i.lt_i$. For simplicity but without loss of generality, we assume the penalty to be linearly proportional to the volume of the rejected task. The *penalty ratio* is the ratio between the total volume of rejected tasks ($\sum\limits_{\forall t_i \in REJECT} v_i(t_i)$) and the total volume of all tasks ($\sum\limits_{\forall t_i \in TS} v_i(t_i)$). When a task is rejected, the total free area in the reconfigurable device is called the wasted area. The *wasted area ratio* is the ratio between the wasted area and the total area of the reconfigurable device. Generally, algorithms with a higher placement quality will exhibit a lower penalty and wasted area ratios.

We evaluated the QC algorithm using real hardware tasks on a real FPGA. We use a benchmark set from [5] (e.g. MDCT, matrix multiplication, hamming code, sorting, FIR, ADPCM, etc) and use the DWARV [4] C-to-VHDL compiler to translate the benchmarks to VHDL. The benchmarks are synthesized with the Xilinx ISE 8.2.01i_PR_5 tools targetting Virtex-4 XC4VLX200 device with 116 columns and 192 rows of reconfigurable units. From these hardware implementations, we obtain the required resources, the reconfiguration times and the execution times of the hardware tasks. Some examples of implemented hardware tasks are shown in Table I. For example, the area A_i for functionPOWER obtained after synthesis is 444 CLBs. In [3], one Virtex-4 row has a height of 16 CLBs. By choosing two rows for implementing this function, we obtain h_i = 2x16 = 32 CLBs and $w_i = \lceil A_i/h_i \rceil = \lceil 444/32 \rceil = 14$ CLBs. The function needs 37 cycles with 11.671 ns clock period (85.68 MHz). Hence, we estimate the execution time of 100 back-to-back operations to be et_i = 37x11.671x100 = 43183 ns. There are 22 frames per column and each frame contains 1312 bits. Therefore one column needs 22x1312 = 28864 bits. Since the function needs 14 CLBs in 2 rows (32 CLBs), we obtain a bitstream with 14x2x28864 = 808192 bits. Since ICAP can send 32 bits per clock cycle at 100 MHz, we estimate the reconfiguration time $rt_i = 808192\text{x}10/32 = 252560$ ns. In the simulation, we assume that the life-time lt_i is the sum of reconfiguration time rt_i and execution time et_i. The hardware tasks are selected randomly from 37 implemented hardware tasks. Every task set consists of 100 tasks, each of which has a life-time and task size. Since we target runtime dynamic multitasking multiuser systems which hardware tasks can arrive any time, the arrival periods of hardware tasks are randomly generated between 10 μs to 20 μs, 20 μs to 30 μs, and 30 μs to 40 μs.

TABLE I
Some examples of implemented hardware tasks(et_i for 100 operations, rt_i on 100 MHz)

No.	Hardware Tasks	$w_i(CLBs)$	$h_i(CLBs)$	$et_i(ns)$	$rt_i(ns)$
1	functionPOWER	14	32	43183	252560
2	adpcm_decode	10	32	770302	180400
3	adpcm_encode	10	32	1031213	180400
4	FIR	33	32	1565980	595320
5	mdct_bitreverse	32	64	449412	1136520
6	mmul	25	64	57278	892980

Because our algorithm is a first fit (FF) heuristic algorithm that can find placements for arrival tasks very fast, we compare the algorithm only with FF heuristic algorithms which are faster than best fit (BF) heuristic algorithms. To fairly evaluate the algorithms, we use the version of Bazargan's algorithm with the best placement quality, i.e. using the FF heuristic for choosing non-overlapping empty rectangles (NERs) and Shorter Segment (SSEG) heuristic for splitting, as mentioned in [1]. In addition, we also compare the proposed algorithm to Inteligent Merging (IM) algorithm (the faster modified version of Bazargan's algorithm). Results were obtained using the aforementioned discrete-time simulation framework and by comparing the following algorithms: Bazargan's (BFFSSEG)

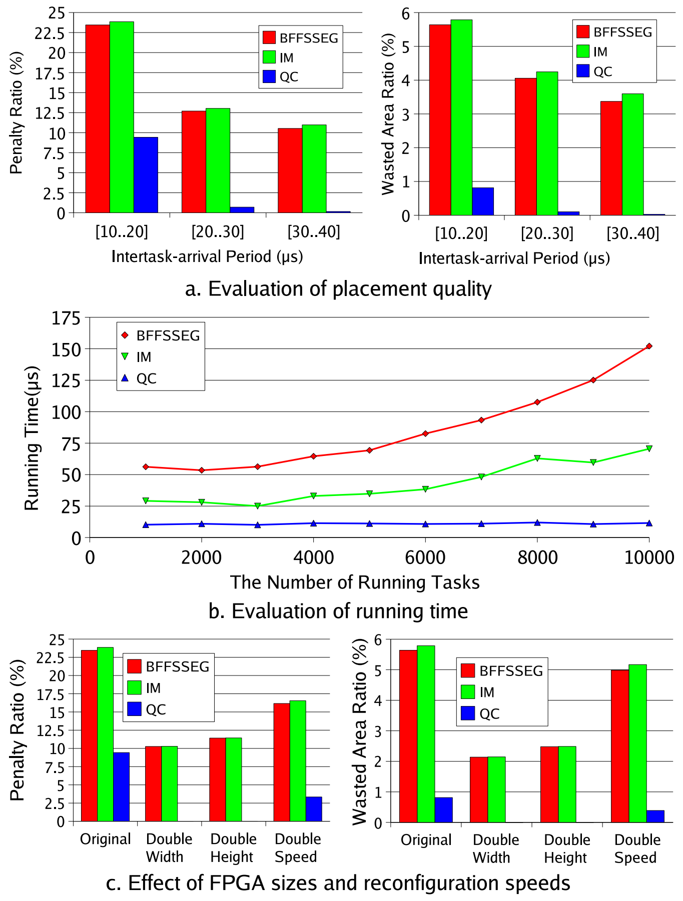


Fig. 4. Evaluation with real hardware tasks

algorithm [1], Intelligent Merging (IM) algorithm [2], and Quad-Corner (QC) algorithm. Average numbers are obtained by running the algorithms 10000 times for every task set.

A longer inter-task arrival period creates more possibilities for additional running tasks to be finished before the arrival of new tasks. As a consequence, the penalty and wasted area are reduced as the increasing of inter-task arrival period.

Due to its on-demand merging, the IM algorithm runs faster than the Bazargan's algorithm with almost similar penalty ratio and wasted area ratio. The Bazargan's and IM algorithms do not perform well because of splitting and fragmentation problems. Figure 4a shows that the QC reduces 78 % penalty and 93 % wasted area of the other related approaches on average by solving above problems.

The increase of the total number of running tasks creates more fragmentation of the free area. As presented earlier, the algorithms that use splitting and merging (Bazargan's and IM algorithms) in managing free area need to merge free area for accommodating arrival tasks. Therefore these algorithms need more merging operations by the increasing of the number of running tasks. As a consequence, the algorithms (exclude the QC) run slower when the number of running tasks increases. Therefore, QC is more scalable in terms of runtime overhead than other algorithms. By totally avoiding merging and its simplicity, QC not only has better placement quality but also runs faster than the other algorithms.

Besides comparing the performance of the algorithms using an original Virtex-4 device (Original), we also measure the effect of doubling the width (Double Width), the height of FPGA (Double Height), and the reconfigurable speed (Double Speed) as depicted in Figure 4c.

Expanding the size of FPGA reduces the penalty and wasted area. The reason is obvious that the larger the FPGA, the more easily to accommodate hardware tasks. As a consequence, the penalty and wasted area are decreased.

Speeding up the reconfiguration also reduces the wasted area and penalty ratios. The reason is that the faster the reconfiguration affects on less life-time of the tasks. As a result, the penalty and wasted area are dropped since the tasks stay shorter on the FPGA.

The effect of doubling the width of FPGA is more beneficial than doubling the height. The reason is that the width of original FPGA (Virtex-4 XC4VLX200) is smaller than its height. As a consequence, doubling the width of FPGA becomes more efficient.

Doubling the size of the FPGA affects more improvement on placement performance than doubling the speed of reconfiguration. The effect of doubling the speed of reconfiguration depends on the ratio of the reconfiguration time and its execution time. The effect will increase if this ratio increases.

IV. Conclusions

In this paper, we have proposed and evaluated a new strategy for online placement of reconfigurable hardware tasks. The main differences between our algorithm and related art are quad-corner spreading capability and dynamical searching sequences. Spreading hardware tasks to the four corners of the devices, finding placements in the specific places, and grouping tasks in free area based on their sizes are the main key features of our proposed algorithm. Experiments with real hardware tasks on Virtex-4 show that the QC not only has 78 % less penalty and 93 % less wasted area than the existing algorithms on average but also has lower runtime overhead.

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] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In IEEE Design and Test of Computers, vol. 17, pp. 68-83, 2000.

[2] T. Marconi, Y. Lu, K. Bertels, G. Gaydadjiev, Intelligent Merging Online Task Placement Algorithm for Partial Reconfigurable Systems, In DATE, Munich, Germany, pp. 1346-1351, March 2008.

[3] Xilinx, Virtex-4 FPGA Configuration User Guide,UG071,2008.

[4] Y. D. Yankova, K. Bertels, S. Vassiliadis, R. J. Meeuws, and A. Virginia, Automated hdl generation: Comparative evaluation, In ISCAS, pp. 2750-2753, May 2007.

[5] R. J. Meeuws, Y. D. Yankova, K.L.M. Bertels, G. N. Gaydadjiev, S. Vassiliadis, A Quantitative Prediction Model for Hardware/Software Partitioning, In FPL07, pp. 735-739, August 2007.