Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing

Herbert Walder, Christoph Steiger, Marco Platzner
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

Abstract

Partial reconfiguration allows for mapping and executing several tasks on an FPGA during runtime. Multitasking on FPGAs raises a number of questions on the management of the reconfigurable resource which leads to the concept of a reconfigurable operating system. A major aspect of such an operating system is task placement. Online placement methods are required that achieve a high placement quality and lead to efficient implementations.

This paper presents placement methods that rely on efficient algorithms for the partitioning of the reconfigurable resource and a hash matrix data structure to maintain the free space. Given n as the number of currently placed tasks, previously known placers find a feasible location in $O(n)$ time. Our approach is able to find a feasible location in constant time. Additionally, simulations show that our methods improve the placement quality by up to 70%.

1 Introduction

Reconfigurable computing systems usually consist of a host processor and reconfigurable devices such as SRAM-based Field-Programmable Gate Arrays (FPGAs). Such systems map algorithms or parts thereof to a circuit which is configured and executed on the FPGA, enabling the execution of algorithms *in parallel* to operations carried out on the host processor. While early FPGAs were rather limited in their densities and reconfiguration capabilities, todays devices provide millions of gates and enable partial reconfiguration and readback. This allows to configure and execute a circuit onto the device without affecting other, currently running circuits. To express their dynamic nature, such circuits are denoted as hardware *tasks*. An application is basically a collection of tasks which are subject to timing, precedence and resource constraints.

The technique of partial reconfiguration can increase device utilization, but it also leads to complex allocation situations for dynamic task sets. This clearly asks for well-defined system services that help to efficiently design and run applications. Such a set of system services forms a *reconfigurable operating system*. From the designer's point of view, an operating system is an additional layer of abstraction in the design process that hides details of the underlying hardware. The benefits of using the additional layer are increased design productivity, portability and resource utilization. On the other hand, these benefits are paid for by overheads in the required area and computation time.

The decision where a task is mapped to determines the fragmentation of the reconfigurable surface. A high fragmentation can lead to the undesirable situation that a task cannot be placed although there would be sufficient area available. A service is needed which organizes the placement of tasks in order to avoid such situations. This work focuses on methods for the online placement of tasks on FPGAs. The objectives are a high placement quality and an efficient implementation of the placement algorithm. Recently, a placer was described by Bazargan et al. [1] that finds a feasible placement for a task in $O(n)$ time, where n is the number of already placed tasks. In comparison, the contributions of this paper are the development of

- a hash matrix approach that finds a feasible placement in constant time, and
- partitioning methods which improve the placement quality by up to 70% compared to [1]

Section 2 reviews related work. Section 3 states the problem modeling. The Sections 4 and 5 discuss the free space partitioning and the fast task placement, respectively. Section 6 presents simulation results.

2 Related Work

Current research dealing with the task placement problem can be subdivided into two categories: offline and online placement. In an offline scenario, one can afford to spend time to derive optimal or near-optimal solutions. A substantial body of work has been done in offline optimization. For example, [2] presents the 3D placement of tasks in time and space dimensions for partially reconfigurable devices.

For online placement, the main reference is [1]. The authors investigated efficient data structures and algorithms for fast online task placement and conducted simulation experiments for variants of first fit, best fit and bottom left

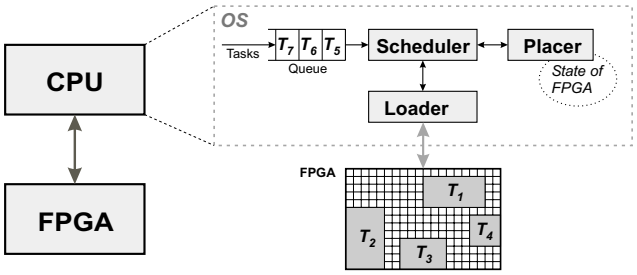

Figure 1. System model and OS modules

bin-packing algorithms. In [3] [4], a different approach is taken to tackle the fragmentation problem on partially reconfigurable FPGAs. Task rearrangements are performed by techniques denoted as local repacking and ordered compaction. In [5] it is proposed to place non-rectangular tasks such that a given fragmentation metrics is minimized. Furthermore, a task's shape may be changed – according to well-defined rules – in order to facilitate task placement. A one-dimensional task placement problem is discussed in [6]. Task relocations and transforms to reduce fragmentation are also proposed in [7]. Task transforms consist of a series of rotation and flip operations. The authors also propose a novel FPGA architecture that supports efficient row-wise relocation.

3 Problem Modeling

This section describes the system, task, and FPGA models assumed in this paper. The assumptions and simplifications made are in accordance with related work in this area [3] [1] [2].

3.1 System Model

We consider the system shown in Figure 1, consisting of a CPU and a partially reconfigurable FPGA. The FPGA provides reconfigurable logic resources, organized in a two-dimensional array of configurable logic blocks (CLBs). The CPU runs operating system functions that manage the reconfigurable system resources.

Arriving tasks are stored in a queue until they get placed and executed on the FPGA. The scheduler decides which of the queued tasks should be loaded and executed next and calls the placer to find a feasible location. The placer manages the free space on the FPGA and tries to find feasible placements for tasks. To that end, the placer module has to maintain the FPGA state. Whenever the placer finds a feasible location for a task, the loader conducts all steps necessary for configuring the task onto the FPGA and starting its execution. On task completion, all reconfigurable resources occupied by the task are freed.

In this work, the following system characteristics hold:

- *Application models:* The system runs applications that may consist of an arbitrary number of tasks. There are no precedence constraints between the tasks. The execution times of the tasks are a-priori unknown. This reflects a general-purpose application scenario.
- *Online scenario:* This work deals with online scenarios, i.e., the operating system does not know in advance at what time tasks arrive and what their properties will be. When a task arrives, the operating system sees the task area and shape but not the task execution time.
- *No preemption:* We assume that tasks cannot be preempted. Once a task is loaded onto the FPGA it runs to completion.

This paper focuses on the placer module of the operating system. Due to the system characteristics, the used scheduler is rather simple. Whenever a new task arrives or an executing one terminates, the scheduler tries to place, load, and execute another task from the queue.

3.2 Task Model

A task is a circuit executable on the FPGA and is described by the following characteristics:

- *Function:* The function captures a task's behavior which is not visible to the operating system.
- *Size / Shape*: Tasks have a certain area requirement, given by the number of CLBs, and a shape. We assume that all tasks have a rectangular shape.
- *Relocatability:* We assume relocatable tasks that can be placed to arbitrary locations on the FPGA. Consequently, a task must not include position-variant resources that restrict the placement.
- *Required cycles*: A task requires a certain number of clock cycles to execute. The actual execution time is determined by the number of cycles and the clock frequency at which the task runs.

3.3 FPGA Model

We make the following assumptions concerning the FPGA:

- *Surface uniformity:* The surface of the FPGA is assumed to be homogeneous. Special resources such as block RAMs are not considered in this work.
- *Routing:* Generally, tasks require intertask communication and I/O. We do not address these issues here but assume that whenever a free area of sufficient size exists enough routing resources are available to fulfill the needs of the placed tasks.

- *Timing:* We do not consider the timing of intertask communication and task I/O routes.
- *Partial reconfigurability:* Arbitrary rectangular subareas of the FPGA can be reconfigured during runtime without affecting other running tasks. That is, we have several tasks on the FPGA executing in parallel.

3.4 Goals

We want to develop placer algorithms and data structures that satisfy two requirements. First, we want to achieve a *high placement quality*. The quality of a placement is expressed by the total execution time and average waiting time of a task set. Formally, a task arrives at time a_i, starts to execute at time s_i and finishes at time f_i. A task's waiting time is given by $w_i = s_i - a_i$. The objectives are:

- If n tasks belong to the same application, we want to minimize the *total execution time* t_{tot} of the task set:

$$t_{tot} = \max_i (f_i) - \min_i (a_i)$$

- For n unrelated tasks which come from different applications, we want to minimize the *average waiting time* $\overline{w}$:

$$\overline{w} = \frac{1}{n} \sum_{i=1}^{n} w_i$$

Second, as we consider online scenarios we are interested in a *fast placer* that shows both a low worst-case runtime complexity and an efficient implementation.

4 Partitioning the Free Space

In an online scenario tasks may arrive and finish execution at any time, leading to complex allocation situations on the FPGA. In order to be able to decide where a newly arrived task can be placed the state of the FPGA, i.e., the free area, must be managed. A straightforward way of managing the free area is to mark each CLB as free or used and to check all possible locations for an arriving task. For an FPGA of size $H \times W$ CLBs, there may be no more than $H \times W$ possible placements.

To reduce this potentially large number of possible locations and increase placement efficiency, Bazargan et al. [1] proposed to base the placement on keeping rectangular areas of free FPGA space. They presented two placement methods. The first method keeps all maximal free rectangles, i.e., rectangles that are not contained in other rectangles. Keeping all maximal free rectangles is optimal in the sense that a feasible location can be found if one exists. On the other hand, the method has to manage $O(n^2)$ rectangles for n placed tasks and task insertion and deletion are difficult to implement. Bazargan's second method sacrifices optimality but is much more efficient as it keeps $O(n)$ non-overlapping rectangles.

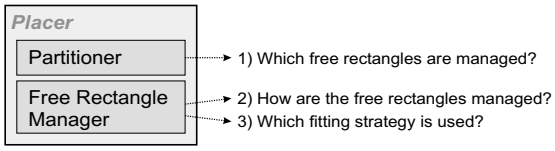

Figure 2. Decomposition of the placer

We can identify three main questions when developing rectangle-based placement algorithms:

1. *Which free rectangles are managed?* We have to decide which set of free rectangles is managed and how the operations task start and task termination are implemented over this set.
2. *How are the free rectangles managed?* We have to choose a data structure that allows for efficient task operations.
3. *What kind of fitting strategy is used?* Generally, there will be more than one possible location for a task. The fitting strategy decides on which one to choose.

We divide the placer module into two submodules as shown in Figure 2. The *partitioner* deals with question 1) and is described in the remainder of this section. The *free rectangle manager* deals with questions 2) and 3) an is elaborated on in Section 5.

4.1 Bazargan's Partitioner

Bazargan's efficient partitioner [1] keeps a number of free rectangles linear in the number of placed tasks. Figure 3 shows the *insert* procedure of Bazargan's partitioner. The placer configures a task T_1 in the bottom-left corner of a free rectangle. The free space splits into two smaller rectangles either vertically or horizontally as shown in Figures 3(b) and 3(c), respectively. To decide on which of the two splits should be performed, Bazargan et al. proposed several heuristics. Because a free rectangle can split into two new rectangles at most, a binary tree is used to represent the FPGA state. The currently free rectangles are the leaves of the tree.

The merge step after the *deletion* of a task basically consists of reverting to the state before the task was inserted. Figure 4 illustrates this. Task T_2 is inserted into rectangle A which splits the residual free space of A into C and D. After the ensuing deletion of task T_2, rectangles C and D are deleted and rectangle A is marked as free again.

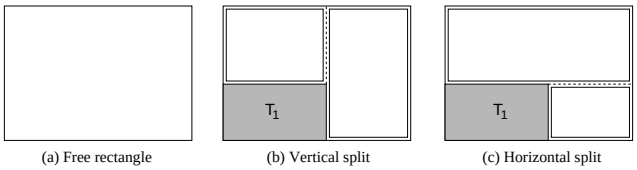

Figure 3. Bazargan's splitting decisions

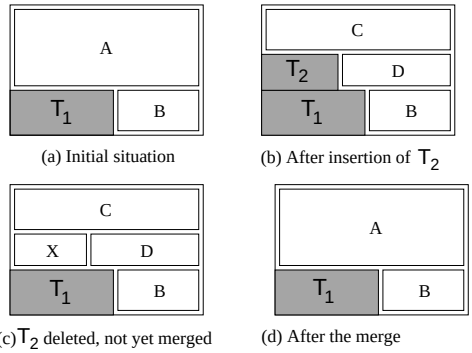

Figure 4. Example of a merge step

4.2 Bazargan's Approach Enhanced

We have developed an enhanced version of Bazargan's partitioner with the same efficiency but improved placement quality. Our enhanced method delays the basic vertical/horizontal split decision and manages overlapping rectangles in a restricted form.

Bazargan et al. use several heuristics to decide whether a free rectangle is split vertically or horizontally on a task insertion. No matter how good such a heuristic is, there is always the possibility of conducting the wrong split. That is, the next task cannot be placed in one of the resulting rectangles due to wrong split decision.

The decisive observation is that the split decision can be *delayed*: whenever a task is inserted into a rectangle, two *overlapping* children rectangles are created as shown in Figure 5(a). The split decision is not made until the next task for one of the two children, A or B, arrives. If the next task is inserted into rectangle A the height of rectangle B is resized such that A and B do not overlap any more. Vice versa, an insertion into B leads to the correction of A's width. Delaying the split decision corresponds to a perfect heuristics: the split decision is taken at a point in time when it is known into which one of the two child rectangles a task is inserted.

Our enhancement of Bazargan's method requires only minor changes in the algorithm. If a task is inserted into rectangle R, we have to check whether R overlaps with its brother rectangle in the binary tree. In such a case we have to resize the width or height of R's brother, respectively. The task deletion procedure is identical to Bazargan's method.

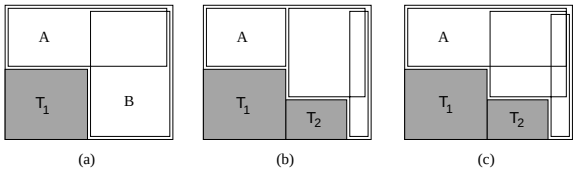

Figure 5. Overlapping child rectangles / motivation for OTF partitioning

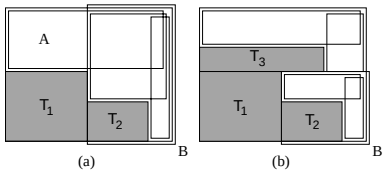

Figure 6. Resizing of brother rectangles

4.3 On-the-fly (OTF) Partitioning

Our OTF partitioner defers the split decision even further. Consider the example shown in Figure 5. Task T_1 has been inserted and two overlapping child rectangles A and B have been created according to our enhanced version of Bazargan's partitioner. (Figure 5(a)). Now task T_2 arrives and is to be placed in rectangle B. The enhanced Bazargan partitioner resizes rectangle A which is shown in Figure 5(b). However, task T_2 does *not* overlap with rectangle A. Therefore, one can leave rectangle A at its original size, getting a better partition of the free space (Figure 5(c)).

The price to be paid is that it might be necessary to resize several rectangles after inserting a new task. Figure 6 illustrates this by extending the example of Figure 5(c). Figure 5(b) shows the result of inserting a task T_3 into rectangle A which overlaps B. The whole subtree of rectangles rooted at B has to be resized, i.e., the height of the rectangles in the subtree is corrected. The implementation of the OTF partitioner differs from the enhanced Bazargan partitioner only in that all rectangles of a subtree might be resized at a later point in time.

4.4 Enhanced OTF Partitioning

The OTF partitioner can further be improved which results in the enhanced OTF method. We have implemented two enhancements:

1. *Resizing rectangles only if necessary*: In the OTF algorithm all rectangles in a subtree are resized if a newly

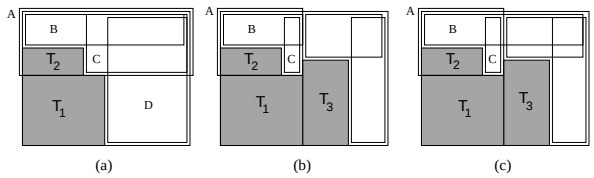

Figure 7. Enhanced OTF: selective resizing

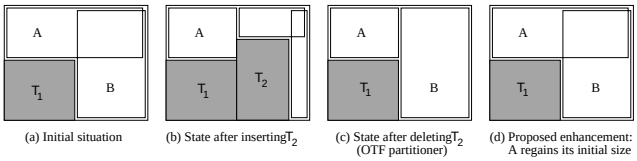

Figure 8. Enhanced OTF: resizing upon deletion

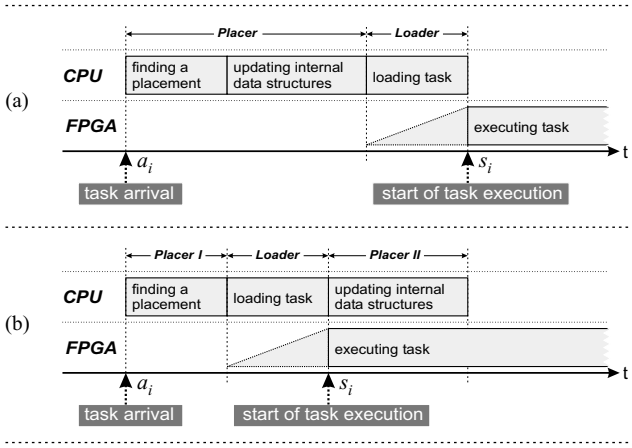

Figure 9. Runtime task placement

placed task overlaps with the root rectangle of the subtree. It might happen that rectangles are resized which do not overlap with the task. An example is shown in Figure 7. Figure 7(a) displays the initial situation occurring when the OTF partitioner inserts the first two tasks T_1 and T_2. Now a new task T_3 is inserted into rectangle D, which leads to the resizing of all rectangles rooted in A. The result of the resizing process is shown in Figure 7(b). Note that rectangle B was resized even though it did not overlap with T_3. Figure 7(c) shows the resulting partition of the free space if B was not resized.

2. *Resizing rectangles upon task deletion*: The OTF partitioner resizes rectangles when new tasks are inserted. If the task which triggered such a resizing is deleted, the resized rectangles do not regain their original size. The enhanced OTF partitioner re-resizes rectangles back to their original size if tasks are deleted. See Figure 8 for an example.

5 Fast Task Placement

Figure 9(a) shows the steps that need to be done when a task arrives. First, the *placer* has to find a location on the FPGA where the task fits in. Second, the *placer* has to update the data structure representing the FPGA state. Finally, the *loader* configures the task onto the FPGA. Since we consider an online scenario, we want to minimize the time it takes to execute the task on the FPGA. The loading time depends on the size of the task and the bandwidth of the FPGA's configuration port. The placement time, however, is largely dependent on the placer's data structures and algorithms. In Bazargan's approach [1] the free rectangles are located at the leaves of the binary tree that represents the FPGA surface. Given a new task, the placer has to search this list for a suitable free rectangle. The updating step involves operations on the binary tree, as described in the previous section.

An important observation shown in Figure 9(b) is that we can start loading a task immediately after finding a feasible placement. The update of the placer's data structure can be delayed and done in parallel to the task execution on the FPGA. In this context, two questions need to be answered:

1. How can we quickly find a suitable free rectangle?
2. Which rectangle should we select if there is more than one suitable?

We have developed placers that maintain a hash matrix additionally to the binary tree. The hash matrix allows to find a suitable rectangle in constant time. Compared to Bazargan et al. we provide the fastest possible method to find a location. The price paid is that we have to spend time for the update of this data structure.

5.1 Hashing

Hashing is a process during which data items are stored in a data structure called *hash table*. A *hash function* maps a key to the entry in the hash table that holds the data item referenced to by the key (see Figure 10).

Given an FPGA of size $H \times W$ CLBs, we define a *hash matrix* as an array ar of size $H \times W$ elements (see Figure 11). A free rectangle of size $a \times b$ is associated with the entry $ar\,[a, b]$ of this array. Every entry consists of a pointer to a list of free rectangles of the corresponding size and a so-called *free pointer*. All free rectangles are stored in the

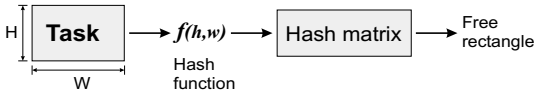

Figure 10. Hashing approach

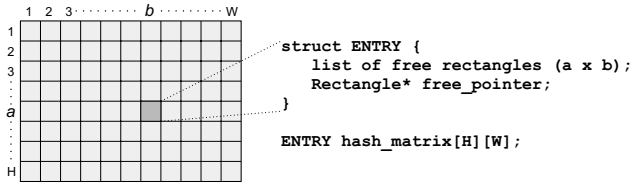

Figure 11. The hash matrix

hash matrix. The following invariant for the free pointer of every entry shall hold:

Invariant The free pointer of entry $ar\,[a, b]$ points to a free rectangle R with $R.height \geq a$ and $R.width \geq b$ according to the fitting strategy.

If the invariant is enforced, finding a free rectangle for a new task is very efficient. Assuming that a newly arrived task has width b and height a, retrieving the suitable free rectangle takes one line of code:

```
return hash_matrix[a][b].free_pointer;
```

This code is executed by the placer module whenever the scheduler asks for a placement. The hashing approach therefore clearly fulfills the timing requirement for an online scenario.

This efficiency comes at the price that all free pointers in the matrix must be kept consistent. Whenever a new task is inserted or deleted, free pointers of some entries need to be updated. Figure 12 shows an example to illustrate this. Initially, 12(a), the hash matrix holds one empty rectangle R_1, reflecting an empty FPGA surface. If a task of size 5×3 is placed, the previously empty rectangle R_1 is deleted and two new free rectangles R_2 and R_3 are inserted into the hash matrix. During this process, all free pointers need to be updated, such that entries belonging to rectangles with sizes up to 7×5 point to R_2 and those with heights $1 \leq H \leq 2$ and widths $6 \leq W \leq 8$ point to R_3. All other entries point to NULL as displayed in Figure 12(b). After task completion, R_2 and R_3 are deleted and R_1 inserted again. At this point of time, the initial situation is re-established (Figure 12(c)).

5.2 Fitting Strategies for the Hashing Approach

If a newly arrived task fits into more than one free rectangle, a fitting strategy is used to choose a rectangle.

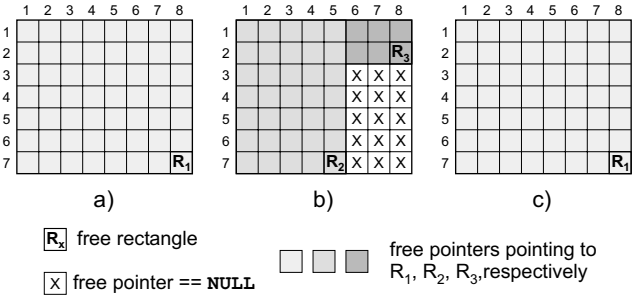

Figure 12. Updating the free pointers

- *Best fit (BF):* BF chooses the free rectangle with the smallest size that can accommodate the task. Intuitively, this strategy tries to keep big rectangles.
- *Worst fit (WF):* WF chooses the free rectangle with the biggest size that can accommodate the task.
- *Best fit with exact fit (BFEF):* Among all rectangles which can accommodate the task, BFEF chooses the smallest rectangle which has exactly the same width or exactly the same height as the task. If no such rectangle is found, the task is placed according to BF.
- *Worst fit with exact fit (WFEF):* Among all rectangles which can accommodate the task, WFEF chooses the biggest rectangle which has exactly the same width or exactly the same height as the task. If no such rectangle is found, the task is placed according to WF.

Figure 13 shows the result of applying the different fitting strategies to an example with rectangles of sizes 3×3, 3×7, 6×2, and 6×5. Entries with the same gray value hold identical free pointers; entries marked with an X contain empty lists of free rectangles.

6 Simulations

In order to evaluate the developed algorithms, we have constructed a discrete time simulation framework which allows to measure the system parameters for randomly generated task sets. All the simulations were conducted for an FPGA with a CLB array of size 96×64, corresponding to Xilinx's Virtex XCV-1000. We have generated task sets in six different classes, varying in the task size. The classes are denoted by C_i and contain tasks of equally distributed size in the interval $[50, i]$ CLBs. The classes are C_{100}, C_{300}, C_{500}, C_{900}, C_{1600} and C_{2700}. These classes have been chosen taking into consideration the area of typical FPGA cores [5]. For every C_i, 50 task sets have been generated with 100 (200 for C_{100}) randomly generated tasks each. Simulation results have been averaged over these 50 task sets.

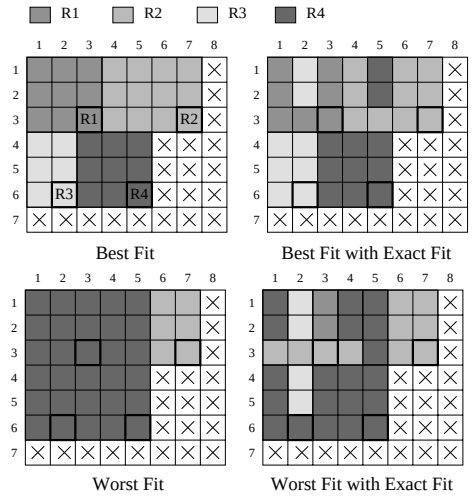

Figure 13. Free pointers according to fitting strategies

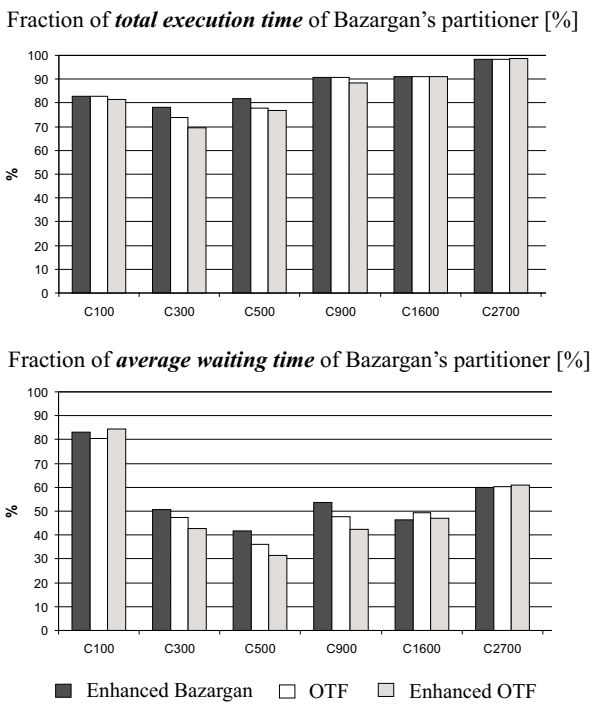

Figure 14. Partitioner performance summary

For all task classes, task computation times are equally distributed in [5,25] time units. Arrival times have been chosen to be equally distributed in the ranges [1,15] to [1,800], depending on the task class. Different arrival time ranges for different C_is make sure that the waiting times of tasks and the system's task load stay within limits that allow a proper analysis of the characteristics and effects of the different placement techniques.

6.1 Evaluation of Partitioning Algorithms

We have simulated all partitioners, combined with all fitting strategies described in previous sections. For each partitioner and C_i, we have selected the fitting strategy yielding the best performance. Figure 14 presents the performance of our three new partitioners compared to the partitioner of Bazargan et al. The figure shows the percentage of the *total execution time* and the *average waiting time* our methods achieve compared to Bazargan's method. The following observations can be made:

- Using our new partitioners, the *total execution time* and the *average waiting time* can be reduced by up to 30% and 70%, respectively. The improvement in total execution time is smaller because this metric depends strongly on the arrival times of the tasks. Tasks with late arrival times diminish any placement improvements achieved before.

- The performance differences are biggest for medium-sized tasks. For small tasks, Bazargan's partitioner splits the free area in a very large and in a very small rectangle. Improved partitioning methods are less effective here as the large rectangle is likely to accommodate any upcoming task. The placement of big tasks leaves only very small rectangles. Placing the next task is then often unsuccessful. Again, the difference between different partitioners is smaller.

6.2 Evaluation of Hash Matrix Operations

To evaluate the runtime complexity of the hash matrix, we have investigated the overhead produced by updating the free pointers. Figure 15 displays the average number of matrix entries scanned per update operation (insertion or deletion of a task) for each fitting strategy. The figure also compares the number of *potential scans* with the number of *real scans*.

If a free rectangle of size $a \times b$ is inserted or deleted in or from the hash matrix, the number of potential scans is $a \cdot b$. In the worst case for the XCV-1000, as many as $96 \times 64 = 6144$ entries need to be scanned. As displayed in Figure 15, even the potential scans are one order of magnitude smaller than the worst case. The fitters BF and BFEF are less expensive than WF and WFEF.

We can observe a remarkable trade-off between the load (number of tasks arriving per time unit) and the number of entries changed in the hash matrix. The higher the load is, the more tasks are on the FPGA and the smaller is the

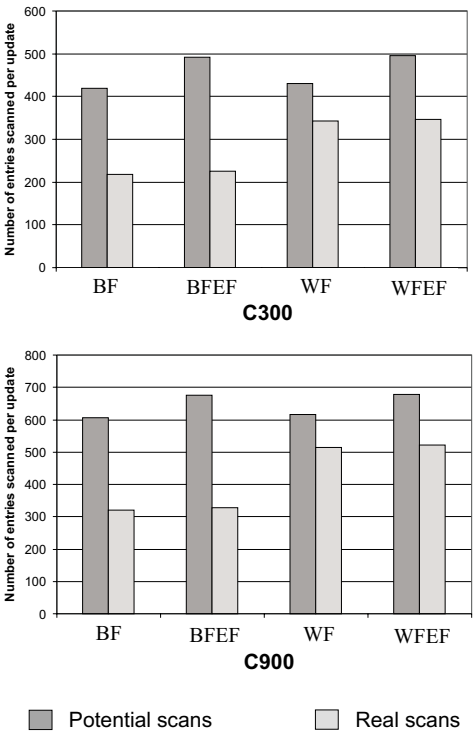

Figure 15. Number of matrix entries scanned

number of matrix entries that have to be scanned. Figure 16 shows the average number of scanned entries per update for different loads, measured for task class C_{500}. The figure displays a significant decrease in the number of scanned entries for higher loads. Both, potential and real scans decrease the same amount. This means that for highly loaded systems where the partitioner is busy, the number of scanned entries decreases and thus, updating the hash matrix is rather cheap. If the load is low on the other hand, the placer module has more time to conduct a more expensive update of the hash matrix.

7 Conclusion and Further Work

In this paper, we presented three newly developed partitioning algorithms based on Bazargan's [1] approach, dealing with empty rectangles. We reported on simulations that show an improvement of up to 70% of the placement quality compared to [1]. We introduced a method based on 2D-hashing to find a feasible task placement with a runtime complexity of $O(1)$, making it suitable for online environments. Further work includes:

- *Repartitioning:* The surface of the FPGA could be repartitioned at some point of time to further improve the placement quality.

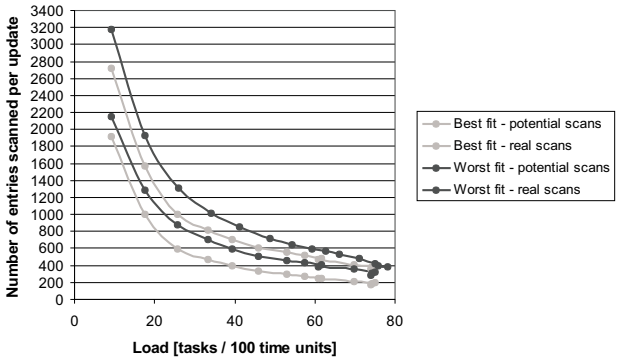

Figure 16. Average number of entries scanned for different loads

- *Relaxing modeling assumptions:* The assumptions made in Section 3 do not fully reflect the requirements of some real-world applications and current FPGA technologies. For example, the techniques presented in this paper must be adapted to deal with task dependencies or FPGA inhomogeneity.

8 Acknowledgements

This work was supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99.

References

[1] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.

[2] Sandor Fekete, Ekkehard Köhler, and Jürgen Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. Design Automation and Test in Europe (DATE)*, pages 658–665, 2001.

[3] Oliver Diessel and Hossam ElGindy. On Scheduling Dynamic FPGA Reconfigurations. In *Proc. 5th Australasian Conference on Parallel and Real-Time Systems (PART)*, pages 191–200, 1998.

[4] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. In *IEE Proceedings on Computers and Digital Techniques*, volume 147, pages 181–188, May 2000.

[5] Herbert Walder and Marco Platzner. Non-preemptive Multitasking on FPGA: Task Placement and Footprint Transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 24–30. CSREA Press, June 2002.

[6] Gordon Brebner and Oliver Diessel. Chip-Based Reconfigurable Task Management. In *Proc. 11th Int'l Workshop on Field Programmable Gate Arrays (FPL)*, pages 182–191, 2001.

[7] Katherine Compton, James Cooley, Stephen Knol, and Scott Hauck. Configuration Relocation and Defragmentation for Reconfigurable Computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE CS Press, April 2001.