A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware

Ali Ahmadinia,Christophe Bobda , and Jürgen Teich

Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Am Weichselgarten 3,
91058 Erlangen, Germany
{ahmadinia, bobda, teich}@cs.fau.de
http://www2.informatik.uni-erlangen.de

Abstract. Recent generations of FPGAs allow run-time partial reconfiguration. To increase the efficacy of reconfigurable computing, multitasking on FPGAs is proposed. One of the challenging problems in multitasking systems is online template placement. In this paper, we describe how existing algorithms work, and propose a new free space manager which is one main part of the placement algorithm. The decision where to place a new module depends on its finishing time mobility. Therefore the proposed algorithm is a combination of scheduling and placement. The simulation results show a better performance against existing methods.

1 Introduction

A reconfigurable computing system is usually composed of a host processor and a reconfigurable device such as an SRAM-based Field-Programmable Gate Array (FPGA)[4]. The host processor can map a code as an executable circuit on the FPGA, which is denoted as a hardware task. With the ability of partial reconfiguration for the new generation of FPGAs, multiple tasks can be configured separately and executed simultaneously. This multitasking and partial reconfiguration of FPGAs increases the device utilization but it also necessitates well thought dynamic task placement and scheduling algorithms [5] Such algorithms strive to use the device area as efficiently as possible as well as reduce total task configuration and running time. But these existing algorithms have not a high performance [1].

Such efficient methods have been developed and perfected in a way such that the hardware tasks are placed on the reconfigurable hardware in a fast manner and that they are furthermore tightly packed to use the available area efficiently. However, most such algorithms are static in nature in the sense that the same placement and scheduling rules apply to every single arriving task and that the entire reconfigurable area is available for the placement of every task. The scope of the present paper hence consists of developing a dynamic task scheduling and placement method on a device divided into slots. More precisely, the FPGA is divided into separate slots, then each of these slots will accommodate only those tasks that end their execution at “nearly the same time”. This 1-D FPGA partitioning as well as the similarity of end times are two

parameters that are dynamically varied during runtime. These parameters must then be controlled by an appropriate function in order to reduce the total execution time and the number of rejected tasks. Finally, relevant statistics are collected and the performance of this newly developed algorithm is then compared experimentally to that of existing ones.
In the subsequent sections previously existing methods and algorithms will be briefly described, the motivation behind the proposed scheduling and 1-D partitioning approach will be explained and the developed algorithm will be described in detail. Finally, comparative results will be presented and analyzed.

2 Online Placement

The problem of packing modules on a chip is similar to the well-studied problem of two-dimensional bin-packing, which is an extension of classical one-dimensional bin-packing [7][8]. The one-dimensional bin-packing problem is similar to placing modules in rows of configurable logic, as done in the standard cell architecture. The two-dimensional bin-packing problem can be used when the operations to be loaded on the modules are rectangles which can be placed anywhere on the chip [1].
In the context of online task placement on a reconfigurable device, the nature of the operations and hence the flow of the program are not known in advance. The configuration of hardware tasks on the FPGA must be done on the fly. To describe the placement problem clearly, we should define our task model:

Definition 1 (Task Characteristics) Given a set of tasks $T = \{ t_1, t_2, \ldots, t_r \}$ such that,

$\forall t_k \in T, \quad t_k = (a_k, e_k, d_k, w_k, h_k)$
a_k = arrival time of task t_k
e_k = execution time of task t_k
d_k = deadline time of task t_k
w_k =width of task t_k
h_k =height of task t_k

This set of tasks must be mapped to a fixed size of FPGA, according to the time and area constraints of tasks. In fact, each task will be mapped to a module which is a partial bitstream. This partial bitstream occupies a determined amount of logic blocks on the device and it has a rectangular shape. Placement algorithms are therefore developed that must determine the manner in which each arriving task is configured. These algorithms must be perfected to, on the one hand, use the available free placement areas efficiently and, on the other hand, execute in a fast manner. However, there most often exists a trade off between these two requirements as fast placement algorithms are usually low-quality ones and those that use the chip area very efficiently compute slowly.
In an online scenario, hardware tasks arrive, are placed on the hardware and end exe-

cution at any possible time. This situation leads to a complex space allocation on the FPGA. In order to determine where the new tasks can be placed, the state of the FPGA or the free area must be managed. This free space management aims to reduce the number of possible locations for the newly arriving tasks and to increase placement efficiency as well. Two such free space management algorithms have been developed in [1] and will be compared to our approach here.
This free space management is the first main part in online placement algorithms. The second part involves fitting the new tasks inside the empty rectangles. Once the free area is managed and the possible locations for the placement of the new task are determined, a choice has to be made at which one of these locations the task will be configured. Multiple such fitting heuristics have been developed in [1]: First Fit, Best Fit and Bottom Left.

2.1 Free Space Management

The KAMER (Keeping All Maximum Empty Rectangles) method has the highest quality of placement as compared to other ones [1]. It is therefore used as the baseline for comparison against other algorithms in terms of the quality of placement that is lost to the benefit of the amount of speed-up that is gained. The KAMER algorithm should hence be described in order to understand why it has such a high placement quality and also why it requires high computation times. In order to decide where the new arriving task should be placed, the KAMER algorithm partitions the empty area on the reconfigurable hardware by keeping a list of empty rectangles. Moreover, these are Maximal Empty Rectangles (MERs), meaning that they are not contained within any other empty rectangle. The arriving task is then placed at the bottom left corner of one of the existing MERs; the choice of the MER depends on the fitting heuristic that is being used. Figure 1 illustrates the case where the empty free space is partitioned into four MERs; their bottom left corners are denoted by an X.

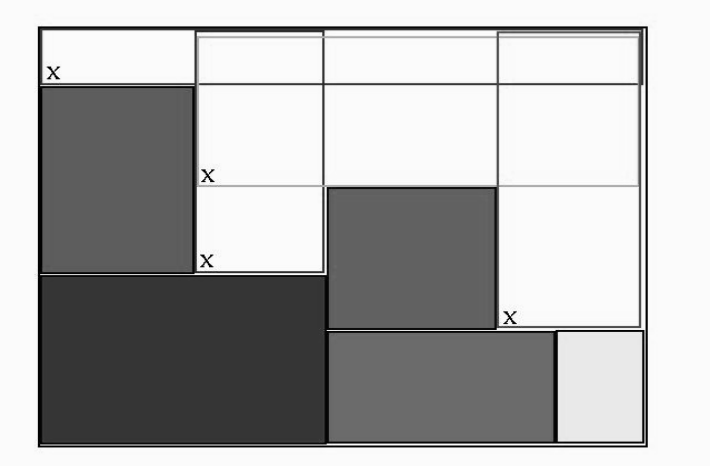

Fig. 1. A free space partition into maximal empty rectangles.

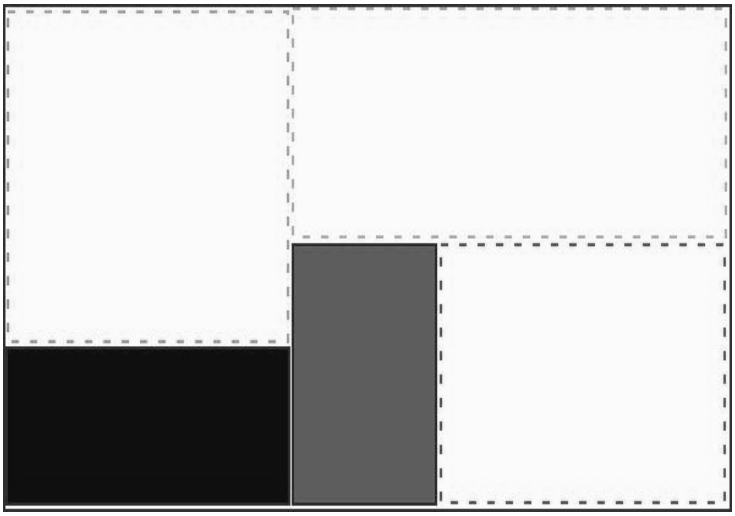

Fig. 2. A free space partition into non-overlapping empty rectangles.

An alternative to the KAMER free space manager is the method that keeps non-overlapping free rectangles. These empty rectangles are not necessarily maximal and hence some quality in placement is lost. The advantage is though that this algorithm executes faster and is more easily implemented. An example of non-overlapping partitioning of the empty region is shown in Figure 2. It should be self evident that in this case of free space management, the empty area can be partitioned in more than one way. Different heuristics can be used on how to choose between different possible non overlapping rectangles.

2.2 Quality

The KAMER placement algorithm is indeed the highest quality method to partition the free space since the rectangles kept in the list and checked for placing the arriving task are maximal and therefore, offer the largest possible area where the new tasks can be accommodated [2]. Keeping all maximum empty rectangles clearly avoids a high fragmentation of the empty space that can lead to the situation where a new task cannot be placed even though there is sufficient free area available.
The reason for quality loss in the keeping non-overlapping rectangles method is that each empty rectangle is contained within one MER. Accordingly, if a task can be placed inside one of these empty rectangles it can also be placed inside the MER that contains it. The reverse is obviously not true. Therefore, this second method for free space management results in a higher fragmentation of the free space and some placement quality is lost.

2.3 Complexity

The KAMER algorithm has to be executed every time a new task is placed on the FPGA as well as every time a task ends its execution and is removed. More precisely, at the moment of the new task's placement, all those MERs that overlap with it must be divided into smaller MERs, and at the moment of a task's removal, the overlapping MERs must be merged into larger ones. As an example, Figure 3 illustrates the partitioning of the free space into five distinct MERs whose bottom left corners are identified by A, B, C, D and E. As the newly arriving task, shown in shaded color, is placed inside MER D, it overlaps with 4 of the 5 existing MERs; B, C, D and E. Each of the latter must then be split into smaller ones. Figure 4 illustrates how MER B is divided into 4 smaller maximal empty rectangles. In the same manner, MER D is split into 2, and MERs C and E are both split into 3 smaller maximum empty rectangles. In this case, the total number of MERs after insertion of the new task increased from 5 to 13. This hence indicates that, in the KAMER algorithm, many MERs must be verified whether they overlap with the new task and furthermore many of them must be divided into smaller MERs. In a similar fashion, after the deletion of a task, a considerable number of MERs must be merged into a few larger ones. Thus, in addition to the increased running time, there is a quadratic space requirement in keeping the number of empty rectangles in a list; this method has to manage $O(n^2)$ rectangles for n placed tasks.

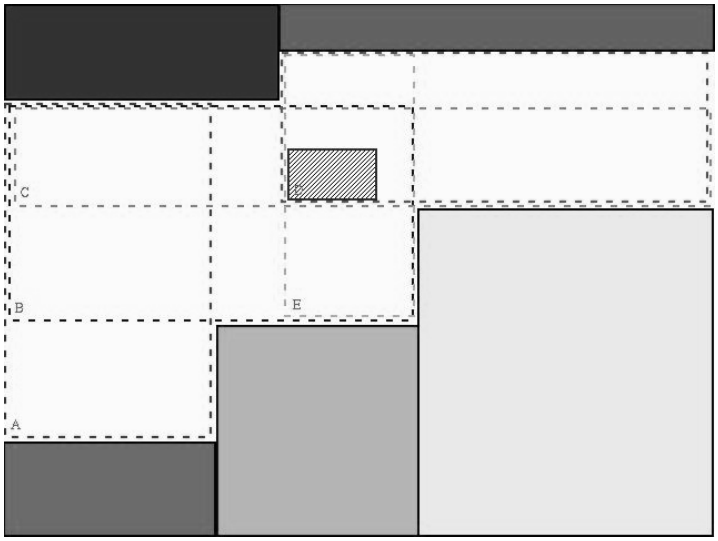

Fig. 3. Placement of an arriving task at the bottom left corner of one of the MERs.

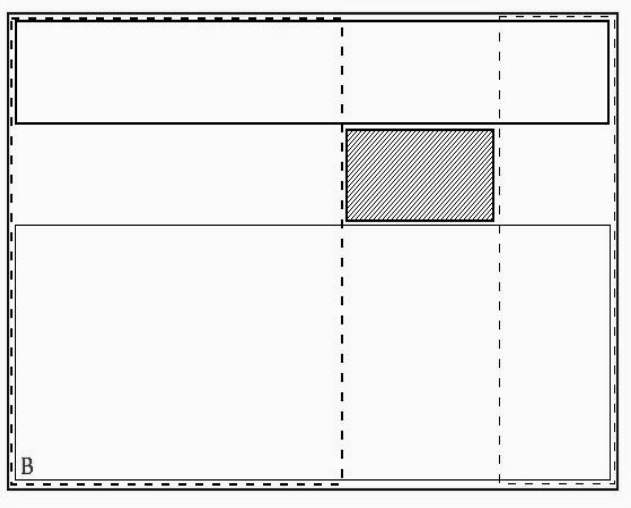

Fig. 4. Changes which are needed in MER B after placing the new module in the bottom left corner of MER D.

It is obvious that the KAMER algorithm, although offering high quality placement, necessitates an important amount of computation and memory, and hence slows down the overall program operation. Consequently, one of the aims of our integrated scheduling and placement algorithm is to execute faster than the KAMER, but by maintaining a certain quality of placement as well.
In the second free space management method, since the empty rectangles are non-overlapping, only the rectangle where the new task is placed should split into two smaller ones. Therefore, we have a O(n) complexity; the number of empty rectangles considered for placing each hardware task is linear in terms of the number of running tasks on the FPGA.

3 An Integrated Scheduling and Placement Algorithm

The aim of this work is to develop an integrated task scheduling and placing algorithm including a 1-D partitioning of the reconfigurable array. In fact a new data structure for management of free space for online placement is developed. Accordingly, the FPGA is divided into slots and the arriving tasks are placed inside one of the slots depending on their execution end time value. Moreover, the width of the slots is to be varied during runtime in order to improve the overall quality of placement. There are two main parameters in this algorithm: The first one determines the closeness of end times for tasks to put in one slot, and the other one defines the width of the area partitioning. A proper function has to be implemented to govern each of these parameters

in order to maximize the quality of task placement. The implemented algorithm will be described in detail and shown to require less memory and computation time than its KAMER counterpart.

3.1 A New Free Space Manager

Unlike in the KAMER algorithm where we have a quadratic memory requirement, our placement algorithm requires linear memory. Instead of maintaining a list of empty rectangles where the arriving task can be placed, we maintain exactly two horizontal lines, i.e. one above and one under the placed running tasks as depicted in Figure 5. For storing the information of each horizontal line, we use a separate linked-list.
In online placement, all of the so far proposed fitting strategies [1][2] place a new arriving task adjacent to the already placed modules, so to minimize the fragmentation. Therefore these two horizontal lines can be determined. As we place new tasks above the *horizontal line_1* or below the *horizontal line_2,* there shouldn't be any considerable free space between these two lines to use the area as efficiently as possible. For example as shown in figure 5, if module 6 is removed earlier than modules 2,3,10 and 11 the area occupied by module 6 will be wasted. To avoid these cases, we suggest placing those tasks beside each other, that they will finish their tasks nearly simultaneously. This task clustering and scheduling will be detailed in the next section.

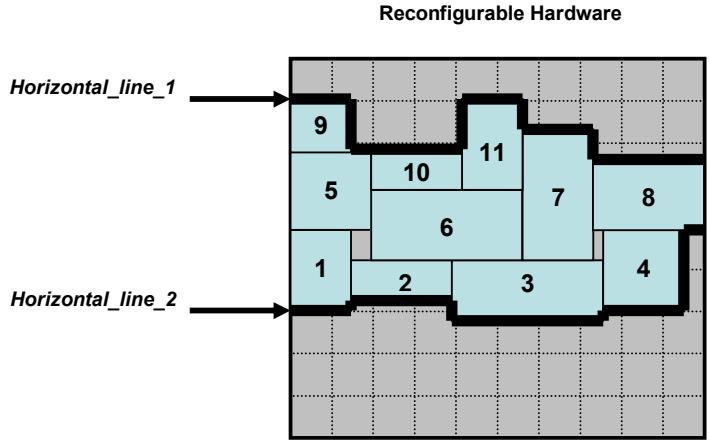

Fig. 5. Using horizontal lines to mange free space.

The placement algorithm is implemented in such a way that, arriving tasks are placed above the currently running tasks as long as there is free space. Once there are no

empty spaces found above the running tasks, the new ones start to be placed below them and so on.
As already mentioned, this implementation requires linear memory. Furthermore, the addition and deletion of tasks involves updating and searching through lists, which is a much faster operation than looking for, merging and dividing maximum empty rectangles. Also, placing the arriving tasks alternatively above and then below the running tasks ensures an efficient use of available area.

3.2 Task Scheduling

As we mentioned before, we need a task clustering to have less fragmentation between the two horizontal lines. For explaining this clustering, first we should define the required specifications of real-time tasks.
Each arriving task $t_k \in T$ is, amongst other parameters, defined by its arrival time a_k and its execution time e_k (definition 1). Hence, if a particular task can be placed on the chip at the time of its arrival, it will end its execution at time $a_k + e_k$. Each task has also a deadline time d_k assigned to it, which is greater than $a_k + e_k$ and sets a limit on how long the task can reside inside the running process. Next we define a mobility interval for each task according to end times. The mobility interval is defined as mobility=[*ASAP_end; ALAP_end*], where $ASAP\_end = a_k + e_k$ is the as soon as possible task end time; and $ALAP\_end = d_k$ is the as late as possible task end time.
Therefore, each task, once placed on the array, will finish its execution at a time belonging to its mobility interval. For clustering tasks, first we should define clusters:

Definition 2 (Clusters or Slots) An FPGA consists of a two-dimensional CLB(Configurable Logic Array) with *m* rows and *n* columns. The columns are partitioned into contiguous regions, which each region is called a *cluster* or *slot*.

The number of slots can be chosen to be different, but in our case, according to the FPGA size and the size of tasks, we have divided the area into three slots. Each task's mobility interval is then used to determine in which cluster the task should be placed. Accordingly, we compute successive end time intervals denoted *end_time1, end_time2, end_time3...* . The details of their computation will be explained in the pseudo code of scheduling.
If a task's mobility interval overlaps with the *end_time1* interval as shown in Figure 6, then this task will be placed inside the first slot, if it overlaps with *end_time2* it will be placed inside the second slot, with *end_time3* inside the third slot, with end_time4 inside the first slot, and so on. This situation is illustrated in Figure 7.
The motivation behind this clustering method, as can be observed in Figure 7, is to have all tasks with similar end times placed next to each other (the number in each module shows that the module belongs to which interval *end_time*). In this way, as tasks "belonging" to the same end time interval end their execution, a large empty space will be created at a precise location. This newly created empty space will then be able to accommodate future, perhaps larger tasks.

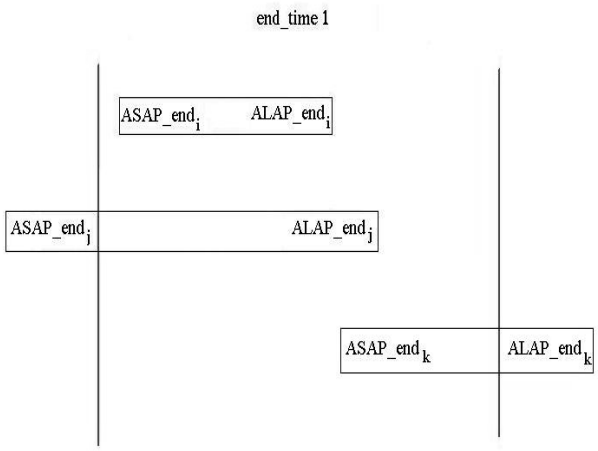

Fig. 6. The tasks with mobility intervals overlapping with the same end time interval.

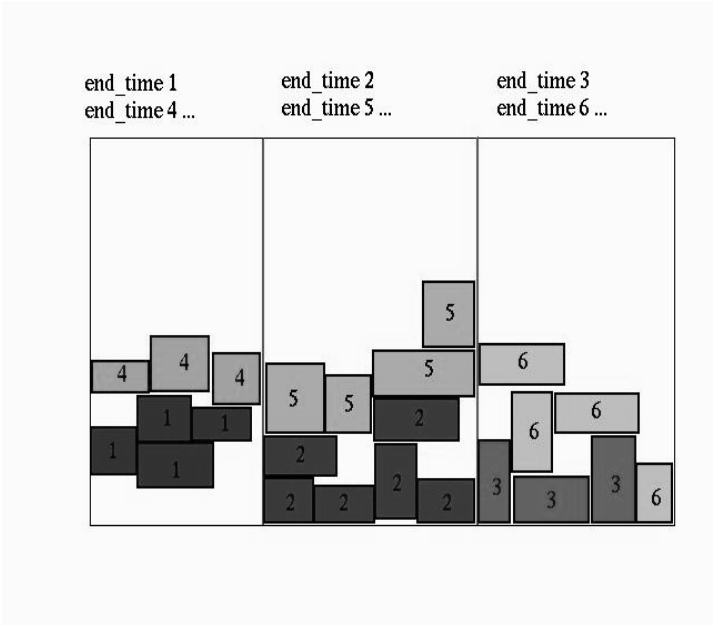

Fig. 7. Placement of tasks inside clusters according to their mobility intervals.

3.3 Optimizing Scheduling

In order to optimize the quality of task placement, or in other words, to reduce the number of rejected tasks, benchmarking had to be performed on how big the successive end time intervals should be. As shown in the pseudo code of computing *end_time* intervals, we divide the *Total_Interval* into three equal ranges. Moreover, an eventual function had to be implemented to vary the ratio of the *end_time* intervals to the *Total_Interval* during runtime. The idea there is that, when an excessive number of tasks are being placed inside a single cluster, the length of the end time intervals should be reduced so that tasks can continue being placed inside the remaining two clusters. Consequently, we define an input rate for each of the clusters as follows:

$$Input \quad rate = \frac{\#\ \ of \quad Tasks}{T} \tag{1}$$

Where *# of Tasks* is the number of tasks placed inside the corresponding cluster during the period *T*. Now, as the input rate of one of the clusters becomes higher than some predetermined threshold value, the length of the corresponding end time intervals should be reduced and kept at that value during some time t. This process was simulated and repeated for a vast range of values for the period T, the threshold and the hold time t. The steps of this scheduling and its optimizing is presented in the following way. Here N is number of slots on the device:

i=0; //number of arrived tasks
k=1;
Max_k=0;
Task_Arriving:
i=i+1;
Min= $ASAP\_end_i$;
Max= $ALAP\_end_i$;
for j=1 to i
{
Min= min(Min , $ASAP\_end_i$);
Max= max(Max , $ALAP\_end_i$);
}
Min= max(Min , Max_k);
Total_Interval=Max - Min;
for s=0 to N-1
$$end\_time(k+s) = (Min + \frac{Total\_Interval}{N} \times s, \frac{Total\_Interval}{N} \times (s+1));$$
if(t mod T=0) // T: period of Time // t: Current time
{
nov=0; // Number of overloaded slots
for s= 1 to N
{

$Task_s = \{t_i \mid t_i \in end\_time(M) \ \ and \ \ M \mod N = s\}$

$Input\_rate(s) = \frac{n(Task_s)}{T}$;

if(Input rate(s) > Threshold)
{
a_s*=1;*
nov=nov+1;
}
else
a_s*=2;*
}
for s=0 to N-1

$$end\_time(k+s) = (Min + \frac{Total\_Interval}{2N - nov} \times \sum_{i=1}^{s} a_i, \frac{Total\_Interval}{2N - nov} \times \sum_{i=1}^{s+1} a_i);$$

}
if (Min > t) // Current time
{
k=k+3;
Max_k*=Max;*
}
Go to ***Task_Arriving***

3.4 Optimizing Partitioning

As a new task arrives, its mobility interval is computed, the overlapping end time interval is determined and the task is assigned to the corresponding cluster. The situation might and will arise where that cluster is full and the task will have to be queued until some tasks within that same cluster end their execution so that the queued task can be placed. However, there might be enough free space in the remaining two clusters to accommodate that queued task. Hence, to improve the quality of the overall algorithm the cluster widths are made to be dynamic and can increase and decrease during runtime when needed. This situation is illustrated in Figure 8.

A proper function had to be found to govern this 1-D partitioning of the reconfigurable hardware. For this purpose, for all the queued tasks waiting to be configured on the device it was counted how many of them are assigned to each cluster. Hence, three variables (*queue1, queue2, queue3*) kept track of the number of tasks in the queue for each of the three clusters. The width of the clusters was then set propor-tionally to the values of these three variables. For example, if during runtime we have the situation where *queue1=4, queue2=2, queue3=2*, the width of the first cluster should be set to half and the widths of the second and third clusters should both be set to one quarter of the entire array width. The widths aren't however changed instantly but rather gradually by some predetermined value at each time unit. This method for the 1-D

space partitioning indeed proved to be the best one in reducing the overall number of rejected tasks.

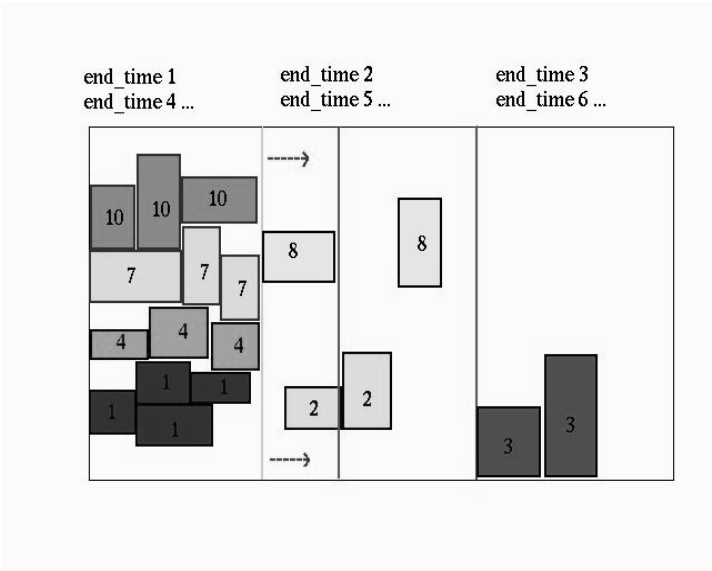

Fig. 8. Dynamic 1-D array space partitioning.

4 Experimental Results

Our cluster based algorithm was compared to the KAMER algorithm in terms of how fast they execute on the one hand, and of how many tasks get rejected on the other. Since the main idea was to compare these two performance parameters, the generation of tasks was kept as simple as possible. Late arriving tasks were not taken into account, only one new task arrives at each clock cycle and once placed, a task's execution cannot be aborted. Also, at every time unit or clock cycle, the algorithm tries to place the new arriving task on the device, checks for tasks that ended their execu-tion so they can be removed and finally all queued tasks are checked for placement. All simulations were performed for a chip size 80x120, a 2-dimensional CLB array, corresponding to the Xilinx XCV2000E device.

In order to evaluate the improvement in the overall computation time of our algorithm as compared to the KAMER, we simulated the placement of 1000 tasks and measured the time in milliseconds both programs took to execute. This was done for different task sizes and shapes, precisely for tasks with width and height uniformly distributed in the intervals [10, 25], [15, 20] and [5, 35]. For each task size range, the

simulation was repeated 50 times and the overall average of execution times was computed. The obtained results are summarized by the graph in Figure 9. For the different task sizes we observe an improvement of 15 to 20 percent as compared to the execution time of the KAMER algorithm. This can be observed in Figure 10, where our algorithm's execution time is presented as a fraction of the time the KAMER algorithm takes to execute.

For optimizing scheduling, the conclusion was that, by varying the width of end time intervals, slightly fewer tasks were being rejected than in the case where the width was just held constant at a single value. In fact, the best performance was observed when the length of the end time intervals was set to be distributed uniformly in the mobility interval range.

The percent of rejected tasks in KAMER was 15.5% and in our cluster-based method was 16.2%. Because, in the KAMER algorithm where the entire chip area is available for all tasks to be placed, tasks with similar end times are most often separated from each other. Once these tasks end their execution, small empty spaces that are distant from each other are created and although there might be enough total free space to accommodate a new task, it might get rejected since not being able to be placed in either of those free locations.

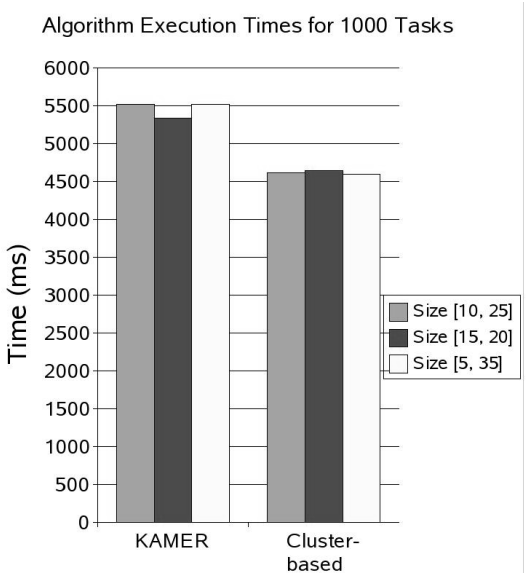

Fig. 9. Algorithm execution times as an average of 50 measurements.

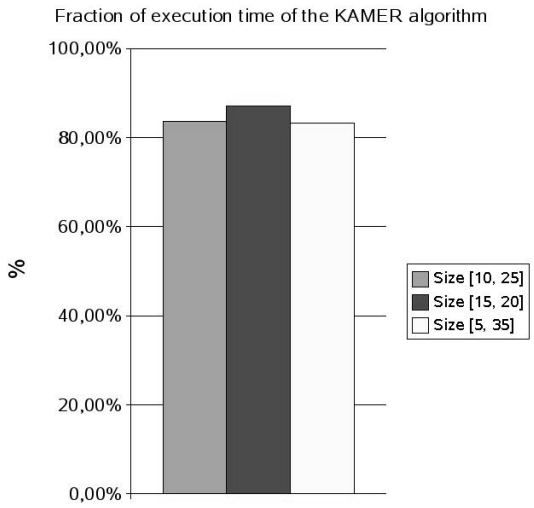

Fig. 10. Fraction of execution time of the KAMER algorithm.

5 Conclusion

In this paper we have discussed existing online placement techniques for reconfigurable FPGA. We suggested a new dynamic task scheduling and placement method. We have conducted experiments to evaluate our algorithm and previous one. We reported on simulations that show an improvement of up to 20% on the placement performance compared to [1]. Also, the quality of placement in this method is comparable to KAMER method and it has nearly the same percent of rejected tasks as KAMER method.

Concerning further work, we plan to develop an online scheduling algorithm to minimize task rejections, and take into consider the dependencies between tasks. Also we intend to investigate more on online placement scenario and make a competitive analysis with optimal offline version of placement [6].

References

1. Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In IEEE Design and Test of Computers, volume17, pages 68-83, 2000.
2. Ali Ahmadinia, and Jürgen Teich. Speeding up Online Placement for XILINX FPGAs by Reducing Configuration Overhead. To appear in Proceedings of 12th IFIP VLSI-SOC, December 2003.
3. Herbert Walder, Christoph Steiger, and Marco. Platzner. Fast Online Task Placement on

FPGAs: Free Space Partitioning and 2-D Hashing. In Proceedings of the 17th Inter-national Parallel and Distributed Processing Symposium (IPDPS) / Reconfigurable Architectures Workshop (RAW). IEEE-CS Press, April 2003.
4. Grant Wigley, and David Kearney, ResearchIssues in Operating Systems for Reconfigur able Computing, In Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA). CSREA Press, Las Vegas USA, June 2002.
5. Oliver Diessel and Hossam ElGindy, On scheduling dynamic FPGA reconfigurations, In Kenneth A Hawick and Heath A James, eds, Proceedings of the Fifth Australasian Conference on Parallel and Real-Time Systems (PART'98) , pp. 191 - 200, Singapore, 1998. Springer-Verlag.
6. Sándor Fekete, Ekkehard Köhler, and Jürgen Teich, Optimal FPGA Module Placement with Temporal Precedence Constraints, In Proc. of Design Automation and Test in Europe , IEEE-CS Press, Munich Germany, 2001, pp. 658-665.
7. E.G Coffman, M.R. Garey, and D.S. Johnson, Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, Approximation algorithms for NP-hard problems, pages 46-93. PWS Publishing, Boston, 1996.
8. E. G. Coffman Jr., and P. W. Shor, Packings in Two Dimensions: Asymptotic AverageCase Analysis of Algorithms, Algorithmica, 9(3):253--277, March 1993.