

Configuration Reusing in On-Line Task Scheduling for Reconfigurable Computing Systems

Maisam Mansub Bassiri and Hadi Shahriar Shahhoseini*

Electrical Engineering Department, Iran University of Science and Technology, Narmak, Tehran, Iran

E-mail: basiri@ee.iust.ac.ir; hshsh@iust.ac.ir

Received March 31, 2010; revised March 19, 2011.

Abstract Reconfigurable computing systems can be reconfigured at runtime and support partial reconfigurability which makes us able to execute tasks in a true multitasking manner. To manage such systems at runtime, a reconfigurable operating system is needed. The main part of this operating system is resource management unit which performs on-line scheduling and placement of hardware tasks at runtime. Reconfiguration overhead is an important obstacle that limits the performance of on-line scheduling algorithms in reconfigurable computing systems and increases the overall execution time. Configuration reusing (task reusing) can decrease reconfiguration overhead considerably, particularly in periodic applications or the applications in which the probability of tasks recurrence is high. In this paper, we present a technique called reusing-based scheduling (RBS), for on-line scheduling and placement in which configuration reusing is considered as a main characteristic in order to reduce reconfiguration overhead and decrease total execution time of the tasks. Several experiments have been conducted on the proposed algorithm. Obtained results show considerable improvement in overall execution time of the tasks.

Keywords reconfigurable computing, on-line scheduling, configuration reusing, RPU partitioning, replacement management, probability of recurrence

1 Introduction

Reconfigurable devices such as modern FPGAs can be reconfigured at runtime and support partial reconfigurability which makes us able to change the configuration of part of FPGA dynamically without disturbing other parts. This feature leads to high flexibility along with high performance in computing systems. Reconfigurable computing (RC) systems comprise one or several reconfigurable processing unit(s) (RPU) along with a CPU^[1]. RC systems allow us to execute tasks in a true multitasking manner.

While using RC systems can increase the computing performance, it also leads to complex allocation situations for dynamic task sets. This clearly asks for well-defined system services that help to efficiently design applications. Such a set of system services forms a reconfigurable operating system (OS)^[2-3]. From the designer's point of view, an operating system is an abstraction layer in design process that hides the details of underlying hardware. The benefits of using this abstraction layer are increased design productivity, portability and resource utilization. Of course these benefits

are paid by area overhead and computation time overhead. This work deals with the resource management part of a reconfigurable OS. In particular, we focus on the problem of on-line scheduling the tasks to the RPU in an RC system. In this paper, we propose an on-line scheduling and placement algorithm which exploits resource reusing (task reusing) and reduces reconfiguration overhead of the tasks to improve the total execution time of the application.

In the rest of this paper, Section 2 discusses the modeling of RPU and tasks. Section 3 reviews the related work. In Section 4 we present our algorithm, called reusing-based scheduling algorithm (RBS). In Section 5, the time complexity of the proposed method (RBS) is presented. Section 6 presents the experimental evaluation of the algorithm. Finally, Section 7 concludes the paper.

2 Problem Modeling

2.1 RPU and Task Models

A task is a function synthesized to digital circuit and can be programmed onto the reconfigurable device

Regular Paper

Supported by a grant from Iran Telecommunication Research Center.

*Corresponding Author

©2011 Springer Science + Business Media, LLC & Science Press, China

(RPU). A task has a size and a shape. The size gives the area requirement of the task in reconfigurable units. We assume all task shapes to be rectangle and the execution time of tasks is known in advance. Generally, each task is defined as a 6-tuple $T_i = (w_i, h_i, e_i, a_i, d_i, r_i)$ where w_i , h_i , e_i , a_i , d_i and r_i denote width, height, execution time, arrival time, deadline and reconfiguration time of the task, respectively.

Tasks can be real-time or non real-time. Deadline (d_i) is defined only for real-time tasks and determines the latest finish time of the task. Also, the tasks can be dependent or independent. Dependent tasks may be represented as a directed task graph. In this work we focus on soft real-time and independent tasks.

Also, the task execution can be preemptive or non-preemptive. Although preemption is a very useful technique that yields efficient and fast on-line scheduling algorithm, we believe that it is too expensive for current reconfiguration technology, because preemption and requires heavy (large) context switches and need additional external memory. Hence, in this work we focus on a non-preemptive task model, i.e., once a task is loaded onto the device, it runs to completion.

The mapping of tasks to an RPU strongly depends on the area model of RPU. Therefore it is necessary to describe our RPU area model. In general, there are four area models. The flexible 2D area model (see Fig.1(a)) is allowed to allocate rectangular tasks anywhere on the 2D reconfigurable surface. This model has been used by many researchers^[4-5]. The advantage of this model is high device utilization but on the other hand high flexibility leads to difficult scheduling and placement algorithms. Another model is partitioned 2D model (see Fig.1(b)) where the reconfigurable surface is split into a statically-fixed number of allocation sites, so called blocks^[6]. Each block can accommodate at most one

task at a time. Such a model facilitates the scheduling and placement but causes internal fragmentation. In the flexible 1D area model shown in Fig.1(c), tasks can be allocated anywhere along the horizontal device dimension and the vertical dimension is fixed. This model matches well current FPGA technology that is partially reconfigurable in vertical chip-spanning columns^[7]. This model leads to low complexity placement problem. On the other hand, it suffers from both internal and external fragmentation. Finally, there is a partitioned 1D area model as shown in Fig.1(d). This model combines the characteristics of a partitioned model and 1D model. Again the disadvantage lies in the potentially high internal fragmentation.

In this paper, we address the problem of on-line scheduling of real-time tasks to the 1D area model. The tasks have known execution time and there is no dependency among them. Such assumptions are the same as used in related research work. There is a large variety of applications which can satisfy these assumptions and exploit our proposed algorithm including: reconfigurable co-processor implementation, image and video processing, cryptography, telecommunication and neural network implementation.

2.2 Scheduling and Placement Problem Definition

In this subsection, we define the problem of scheduling and placement. As stated earlier, in this work we use 1D area model for device area. Therefore the location of each task T_i on the device surface is indexed with x_i which is the horizontal position of bottom-left of rectangular task (T_i). We can define the placement and scheduling for a task as follows:

Definition 1. A placement $P(T_i, T_c)$ for a task T_i and a set of currently placed tasks T_c , is a placement $\langle x_i \rangle$ that satisfies the following constraints in 1D area model:

$$\begin{aligned} x_i + w_i &\leq W \\ h_i &\leq H \\ \forall T_j \in T_c : (x_i + w_i &\leq x_j) \vee (x_j + w_j \leq x_i). \end{aligned}$$

H and W are height and width of the RPU surface, respectively. In fact, the hardware tasks must be placed in such a way that they do not overlap with the device boundaries and other currently placed tasks.

Definition 2. Scheduling assigns a starting time s_i to each task $T_i \in T$ such that:

$$\begin{aligned} \forall T_j \in T, \quad T_j &\neq T_i : \\ (x_i + w_i &\leq x_j) \vee (x_j + w_j \leq x_i) \vee \\ (s_i + e_i &\leq s_j) \vee (s_j + e_j \leq s_i) \end{aligned}$$

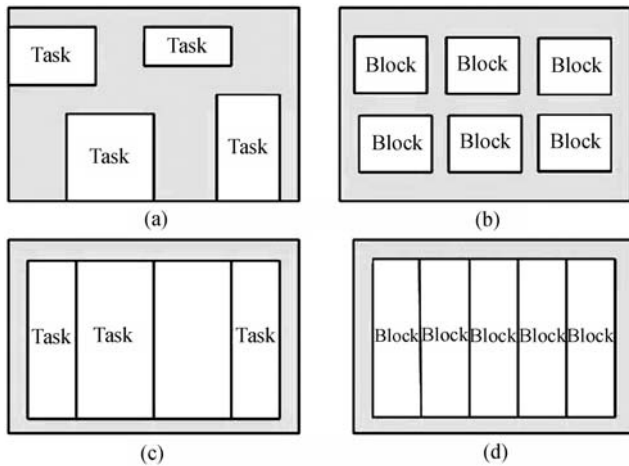


Fig.1. Reconfigurable area models. (a) Flexible 2D. (b) Partitioned 2D. (c) Flexible 1D. (d) Partitioned 1D.

$$s_i + e_i \leq d_i.$$

In an on-line scenario, new tasks arrive during the system's runtime. For each arriving task T_i , scheduler has to find feasible start time. If scheduler and placer cannot find a feasible placement and start time for the arrived task that satisfies the aforementioned constraints, the arrived task will be rejected. In such real-time systems, it is assumed that when a task is rejected, it is referred to the resource out of the system for execution. Two main goals are considered for on-line scheduling and placement in RC systems. One of them is to minimize the task rejection ratio (TRR) in the system. Task rejection ratio is the ratio of the number of rejected tasks to the total number of arrived tasks. Another main goal in on-line scheduling and placement is to decrease the overall execution time of the tasks.

Regarding above descriptions, we state the problem in this paper as follows:

On-line scheduling and placement of independent tasks in the RPU with 1D area model. The main goals are minimizing the task rejection ratio and overall execution time of the tasks.

2.3 Task Placement

Generally, there exist two different approaches for task placement: scanning and management of free spaces.

The main advantage of the scanning techniques is that they are recognition-complete. This means they are optimal in the sense that they find a feasible placement for a new task if sufficient resources (free reconfigurable units) are available. The drawback of this approach is its high runtime complexity for large device area. Of course, in 1D area model, the runtime complexity of scanning technique is acceptable. In fact, in the worst case, W (the width of the RPU in reconfigurable units) potential placements have to be considered.

In free space management techniques, the list of free spaces is maintained and is updated in each event. In these techniques, for placement of a task, only the free spaces are searched. Some of space management techniques such as MER (maximal empty rectangle) management^[4] are recognition-complete and some of them are not.

In this paper, we use scanning-based placement technique for placing the tasks on the RPU. Generally a placer contains two main modules: status holder module and fitter module. The status holder maintains the status of reconfigurable units. Whenever a task is placed or removed, status holder updates the list of free reconfigurable units. The fitter module selects a

site among the several feasible sites for task placement. Several fitting strategies such as *first-fit* and *best-fit* can be applied in task placement.

In this paper, we present an on-line scheduling and placement algorithm for RC systems which focuses on high configuration reusing (task reusing) in order to decrease the reconfiguration overhead caused by placement of new arriving tasks.

3 Related Work

So far, several important researches have been carried out on scheduling and placement in RC systems. Most of them such as [8-10] have addressed off-line (compile-time) scheduling and placement in RC systems. In off-line scenario, the input task graph is known in advance and there is enough time to find the optimal solutions. For example, Banerjee *et al.* in [8] present a placement-aware algorithm for integrated HW/SW partitioning and scheduling in off-line applications. Also, Noguera *et al.* in [9] present a semi-online algorithm that performs off-line partitioning and on-line scheduling in a multi-context RC system. In other off-line methods such as [10], authors use configuration prefetching in scheduling to mitigate configuration overhead. Also, Huang *et al.* in [11] propose two algorithms which consider data dependency among tasks in a task graph to reduce the number of configurations.

Most of the off-line algorithms use ILP technique or evolutionary algorithms for task scheduling and placement which are not usable in on-line applications due to their high computation time.

Second category of the recent work such as [12-20] address on-line scheduling in RC systems. Some of these studies such as [13] focuses on real-time and preemptive tasks. But most of them focus on real-time and non-preemptive tasks. In [16], two techniques, *horizon* and *stuffing*, are presented by Steiger, Walder and Platzner. Both proposed techniques use reservation list and free space list for scheduling. The stuffing method schedules tasks into arbitrary free rectangles that will exist in the future. But horizon scheduler can only append the tasks to the horizons and it does not guarantee to find a feasible placement even though there is enough free space on the RPU. In *stuffing* method, when a new task T_i arrives, the scheduler starts walking through the task's planning period, simulating all future allocations of the device by mimicking task terminations and placements together with the underlying free space management. On arrival of T_i and the termination of the n -placed tasks, the placer is called to find a feasible interval. If one is found, the scheduler accepts and adds a new reservation for T_i . If no sufficient interval is found, the scheduler proceeds until the latest start

time of T_i .

Zhou *et al.* in [12] present a technique that uses time window to find feasible location for placing on-line real-time tasks. In fact, they have improved the *stuffing* algorithm for scheduling. Ahmadiania *et al.* in [14] present an integrated scheduling and placement algorithm in which the FPGA is divided into slots and the arriving tasks are placed inside one of the slots depending on their execution end time. Moreover the width of the slots is to be varied during runtime in order to improve the quality of placement.

Marconi *et al.* in [17] present *intelligent stuffing* method. They define alignment status of the free space segments and use it to make the correct decision on task placement position in order to maximize the free space area. In another work^[20], Roman *et al.* have proposed a technique in which FPGA area is divided into four partitions of different sizes. Each partition has an associated queue where the hardware manager places each arriving task depending on its size, shape and real-time parameters as deadline. The algorithm may change the queue selection policy, partition strategies and the sizes or the number of partitions at runtime in order to adapt itself to the parameters of arriving tasks.

In [21], Cui *et al.* present a novel fragmentation metric based on MERs that takes into account probability distribution of width and height of future task arrivals instead of just the average value. In addition, they take into account the time axis to minimize the time-averaged area fragmentation (TAAF) during the execution time of the task being placed. They use TAAF to select the best feasible placement among the several placement choices. Dittmann *et al.* in [22] propose an on-line scheduling algorithm which considers simple task reusing in scheduling new arriving tasks.

4 Proposed On-Line Scheduling and Placement Algorithm

As stated in Section 3, most of on-line scheduling and placement algorithms for RC systems which have been presented so far, maintain the list of free spaces on RPU surface and update the list at any task arrival and task termination time^[4,23-25]. In such methods, once the execution of a task finishes, the task is removed from RPU surface and its area is added to free space list. The main drawback of such techniques is that they cannot exploit configuration reusing (task reusing) as much as possible. Fig.2(a) shows an example in which task T'_1 is an instance of T_1 (a copy of T_1) and arrives at time a'_1 . In this situation this task should be reconfigured again on the free area of RPU. In this figure, dotted parts of the tasks (R_i) indicates the reconfiguration time of the tasks. Reconfiguration overhead (reconfiguration

time) is an important obstacle that limits the performance of the RC systems and increases the overall execution time of the tasks^[10]. Configuration reusing (task reusing) can decrease reconfiguration overhead considerably, particularly in periodic applications or applications in which some tasks are repeated frame by frame or packet by packet such as audio and video processing, cryptography, network applications and so on. Fine granularity increases the probability of task repetition in applications.

In our proposed algorithm, called reusing-based scheduling algorithm (RBS), configuration reusing is considered as a main characteristic in order to reduce reconfiguration overhead and improve the total execution time of the tasks.

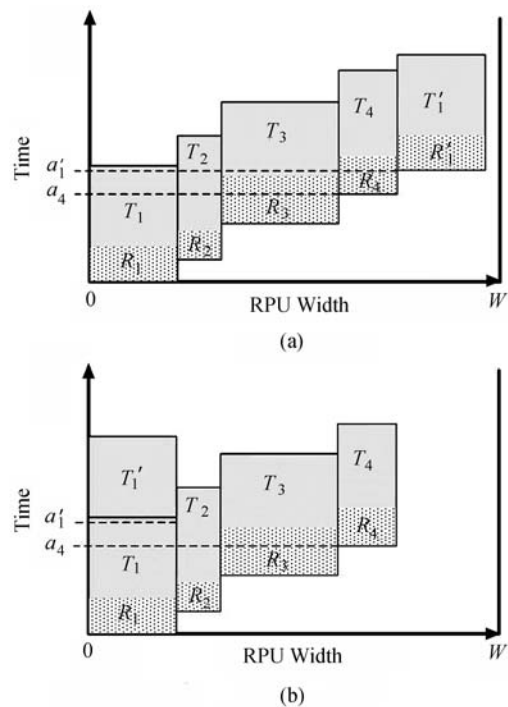


Fig.2. An example of task scheduling and placement. (a) Without reconfiguration reusing. (b) With reconfiguration reusing.

The basic idea of reusing-based scheduling (RBS) method is that some of arrived tasks are not removed from RPU surface after the completion of their execution and they stay on RPU surface as far as possible. Therefore if these completed tasks (finished tasks) recur in the future, we can reuse the existing configuration of the recurring tasks and the reconfiguration overhead will be eliminated. Fig.2(b) shows an example of this situation.

Although maintaining the tasks on the RPU surface increases the probability of task reusing and decreases the reconfiguration overhead, it is not possible to maintain all arrived tasks on the RPU. Therefore we divide

the arriving tasks into two groups: significant tasks and non-significant tasks. We only preserve significant tasks on the RPU after their completion. Non-significant tasks are removed from RPU surface after the completion. In consequence, the existing tasks on the RPU are divided into three categories at any given time:

- 1) *running tasks* which are currently executing on the RPU;
- 2) *scheduled tasks* which have been scheduled but their execution have not been started yet;
- 3) *preserved tasks* which their execution have been completed.

Also, the area of RPU is divided into three categories at any given time:

- 1) *free area* in which there is no task;
- 2) *occupied area* which includes the running tasks and scheduled tasks;
- 3) *preserved area* which includes the preserved tasks.

In Fig.3, the shaded area indicates the preserved area and white area indicates the free area on the RPU.

Preserving some tasks on the RPU surface after their completion and removing other tasks may lead to surface fragmentation on the RPU which will be described in next subsection. In order to prevent RPU surface fragmentation, we partition RPU surface and put together the significant tasks in a partition. Non-significant tasks are placed in another partition. It is necessary to be able to change the size of partitions dynamically (during runtime). This makes us able to maintain the algorithm performance even when the task recurrence is low.

Another point that must be considered is that when the number of preserved tasks on the RPU or the workload (number of arriving tasks per time unit) increases, preserving significant tasks on the RPU surface may increase the probability of rejection of the new arriving tasks because the free area is reduced. In order to prevent task rejection in such situations, we propose a technique for task replacement management which makes us able to replace preserved tasks with newly arrived task. In fact, RBS method decreases the overall execution time of the tasks while preserving the rejection ratio in low value.

According to above description, RBS algorithm includes four main parts: 1) task and RPU partitioning, 2) task scheduling and placement, 3) dynamic partition resizing, 4) task replacement management.

In the rest of this section, we describe each part of RBS method in details.

4.1 Task and RPU Partitioning in RBS Method

As stated before, in RBS method, we divide the

arriving tasks into two groups: significant tasks and non-significant tasks.

We define the significance of a task based on the following parameters:

- 1) Reconfiguration overhead: the ratio of reconfiguration time to execution time of a task (r_i/e_i); this parameter is denoted by O_i .
- 2) The probability of task recurrence in the future; this parameter is denoted by P_i .

Parameter P_i can be calculated using Poisson probability distribution function. According to Poisson distribution function, the probability of arriving k tasks in next time interval Δt is calculated as follows:

$$P_{\Delta t}(k) = \frac{\lambda^k \times e^{-\lambda}}{k!} \quad (1)$$

where, λ is the average number of arrived tasks in past time interval Δt .

Thus, the probability of recurrence, P_i , for task T_i , is the probability of arriving one instance of task T_i in next time interval Δt , and is calculated as follows:

$$P_{\Delta t}(1) = \frac{\lambda_i^1 \times e^{-\lambda_i}}{1} = \lambda_i \times e^{-\lambda_i} = P_i \quad (2)$$

where, λ_i is the average number of arrived instances of task T_i in past time interval Δt .

The significance of a task, SG_i , is a Boolean parameter which is defined as follows:

$$SG_i = (O_i \geq k_1) \wedge (P_i \geq k_2) \quad (3)$$

where k_1 and k_2 are the thresholds between 0 and 1; when the value of SG_i is "1", the task is considered as significant task and vice versa. (3) states that the task is significant only if it has relatively large reconfiguration overhead (O_i) and large probability of recurrence (P_i). Consequently, significant tasks are eligible for preserving on the RPU after their completion. Preserving significant tasks may eliminate the reconfiguration overhead and decrease the total execution time of the tasks.

We define a task list which maintains the history of arriving tasks including: the number of task recurrence along with recurrence time. When a new task arrives at the system, SG_i is calculated for newly arrived task using the task list information.

As stated before, preserving some tasks and removing other tasks after their completion may incur fragmentation on the RPU surface and increase the probability of task rejection in the future, consequently. Fig.3(a) indicates an example in which tasks T_2 and T_5 are significant and the others are non-significant. Task T_6 arrives at time a_6 but it cannot be placed on the RPU due to the free area fragmentation on the RPU surface. Fig.3(b) shows the same example with suitable

task placement policy. As seen, this reduces the area fragmentation and decreases the probability of task rejection.

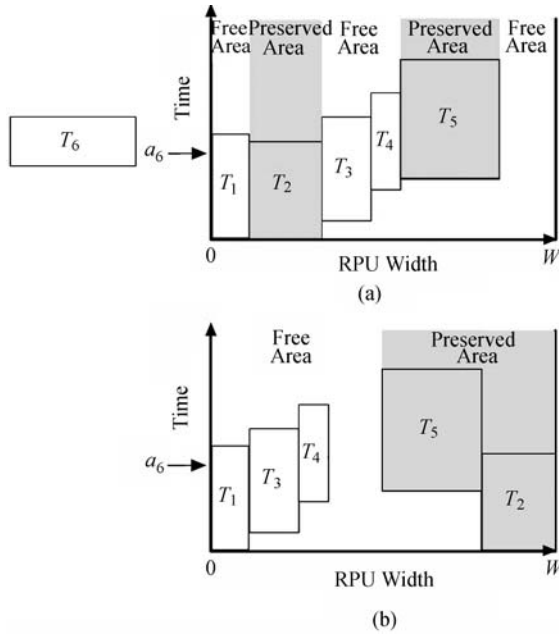


Fig.3. Example of RPU surface fragmentation due to the placement policy. (a) Unsuitable placement policy. (b) Suitable placement policy.

Regarding the above description, we partition the RPU surface into two partitions: P_1 and P_2 . Fig.4 shows a total scheme of this partitioning. In initial partitioning, the width of each partition is defined as follows:

$$\begin{aligned} W_{P_1} &= \alpha \cdot W, & W_{P_2} &= \beta \cdot W \\ 0 < \alpha, \beta < 1, & \alpha + \beta &= 1 \end{aligned} \quad (4)$$

where W_{P_1} and W_{P_2} denote the width of partitions P_1 and P_2 , respectively. Also, α and β are positive coefficients between 0 and 1.

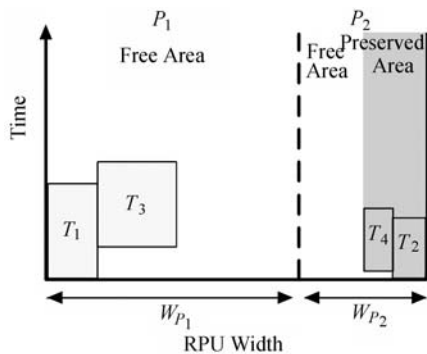


Fig.4. Total scheme of RPU partitioning.

Partition P_1 is dedicated to placing the non-significant tasks and partition P_2 is dedicated to placing the significant tasks.

The boundaries of the partitions are virtual and can be moved during the runtime. In fact, the partitions can be resized dynamically according to the conditions of input tasks. The details of partition resizing are explained in Subsection 4.3.

The main goal of RPU partitioning is that the tasks with similar significance are put together in one partition. This causes that the preserving significant tasks and removing non-significant tasks do not lead to much fragmentation on RPU surface.

4.2 Task Scheduling and Placement in RBS Method

In RBS method, we use scanning-based placement method for task placement on the RPU. The tasks are placed from left to right in partition P_1 . In contrast, the tasks are placed from right to left in partition P_2 . This facilitates partition resizing which will be described in Subsection 4.3.

In what follows, we describe our scheduling and placement strategy for different situations in RBS method.

When a new task arrives at the system, two different situations may occur:

- (a) the newly arrived task (T_i) is non-significant, i.e., $SG_i = "0"$;
- (b) the newly arrived task (T_i) is significant, i.e., $SG_i = "1"$.

In situation (a), following steps are performed respectively to schedule and place the newly arrived task:

1) Free area in partition P_1 is scanned using *stuffing method* (see Section 3) and the new task is placed on the free area provided that enough free area is found on the partition.

2) If there is not enough free area on partition P_1 , then the possibility of partition resizing is examined (checked). If partition resizing is feasible, then partition P_1 is expanded and the new task may be placed in newly emerged area. Details of partition resizing are described in Subsection 4.3.

3) If partition resizing is not allowed (feasible), the newly arrived task will be rejected.

In situation (b), following steps are performed respectively to place the newly arrived task:

1) The scheduler scans the occupied area and preserved area in partition P_2 to find an existing instance of the new task.

2) If there exists a preserved instance of newly arrived task on partition P_2 , the new task will be scheduled on the existing preserved instance and

reconfiguration overhead will be eliminated.

3) If the existing instance of new task is running, then the finish time (f_j) of running instance is calculated according to the start time and execution time of the task ($f_j = s_j + e_j$). Also, the latest start time (LST) of new task is calculated regarding its deadline ($LST_i = d_i - e_i$). If the LST of new task is greater than the finish time of running instance, the new task will be scheduled immediately after the finish time of running instance and reconfiguration time of the new task will be eliminated.

4) If the finish time of running instance is greater than the LST of new task, then new task is placed on the free area of partition P_2 and the running instance will be removed after its completion.

5) If there is not enough free area on the partition P_2 to place new task, the feasibility of partition expansion (resizing) will be examined. If partition expansion is allowed, then the partition is resized and new task may be placed on newly emerged area.

6) If partition expansion is not allowed or new task cannot be accommodated into newly emerged area, one of preserved tasks in partition P_2 is selected to be replaced by newly arrived task. The details of task replacement are explained in Subsection 4.4.

7) If task replacement is not possible, the new task will be rejected.

It must be remembered that the significant tasks are preserved on the RPU surface after the execution completion, but non-significant tasks are removed from RPU surface after the completion. Also, all the lists which maintain occupied, preserved and free areas are updated at each task arrival time and task completion time.

4.3 Dynamic Partition Resizing in RBS Method

When there is not enough free area on a partition to place new task, partition resizing may be a good solution to prevent task rejection. Partition resizing is to expand the size (width) of the partition provided that there is excess free area on adjacent partition. We define a parameter, called *partition utilization* (PU), for each partition as the following:

$$\begin{aligned} PU_i &= \frac{W_{occupied} + W_{preserved}}{W_{P_i}} \\ &= \frac{W_{P_i} - W_{free}}{W_{P_i}} \end{aligned} \quad (5)$$

where $W_{occupied}$, $W_{preserved}$ and W_{free} are total width of occupied area, total width of preserved area and total width of free area on partition P_i , respectively. The

less the value of PU_i , the more there is free area on the partition P_i .

Resizing of partition P_i is allowable (feasible) only if the following constraint is satisfied:

$$PU_k \leq 0.6, \quad P_k : \text{adjacent partition of } P_i. \quad (6)$$

Above constraint states that resizing of partition P_i is allowed only if the adjacent partition of P_i has $PU_k \leq 0.6$. Therefore, if P_i 's neighbor has $PU_i > 0.6$, the resizing of partition P_i will not be allowed. In fact, when a partition expands, the next partitions will be contracted. We limit the maximum amount of expansion for partition P_i to $((1 - PU_k)/2) \times W_{P_k}$, where PU_k and W_{P_k} denote the utility and width of partition P_k which is the neighbor of P_i and satisfies the above constraint. Fig.5 shows an example of partition resizing in which partition P_1 has been expanded.

Partition resizing makes us able to change the virtual boundaries dynamically. This can decrease the probability of task rejection when the workload increases or when the task recurrence is low in an application.

4.4 Task Replacement Management in RBS Method

When there is not preserved instance of newly

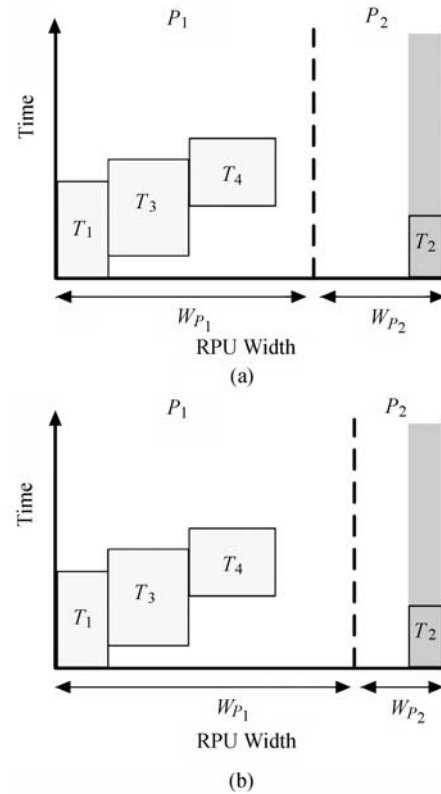


Fig.5. An example of partition resizing. (a) Before expansion of P_1 . (b) After expansion of P_1 .

arrived task on partition P_2 and also there is not enough free space on this partition to place the new task, one preserved task should be selected from preserved area to be replaced by newly arrived task. Two different policies can be used: *best-fit* policy and *least probability of recurrence (LPR)* policy.

In *best-fit* policy, the smallest preserved task which can accommodate newly arrived task is selected for replacement. In fact, *best-fit* policy aims at reducing the fragmentation on RPU surface.

In *LPR* policy, the task which has the smallest probability of recurrence and is large enough to accommodate the new task is selected for replacement. In fact, the task which has the smallest probability of recurrence is the least recently recurred task, because the probability of recurrence, P_i , depends on the number of task recurrence in recent time window (λ). In Subsection 6.2, we compare the effect of using these two replacement policies on overall execution time of the tasks.

In situation that none of preserved tasks can accommodate newly arrived task, two contiguous tasks which have the smallest total width and can accommodate the newly arrived task are selected for replacement by newly arrived task.

5 Complexity Analysis of RBS Algorithm

Generally, the RBS algorithm includes three main steps as follows:

- 1) searching the free space or occupied space in each partition;
- 2) partition expansion for each partition;
- 3) task replacement in partition P_2 .

Since searching the free/occupied space in each partition is performed using the scanning method, the complexity of searching free/occupied space is $O(W_{P_i})$, where W_{P_i} is the width of partition P_i .

Partition expansion for each partition requires the calculation of partition utilization using (5). The complexity of this work is $O(W_{P_i})$, where W_{P_i} is the width of partition P_i .

Task replacement is performed only in partition P_2 and its complexity is $O(n_2)$, where n_2 is the number of preserved tasks in partition P_2 . In the worst case n_2 may be equal to W_{P_2} .

We use W (width of the RPU) instead of W_{P_i} for considering the worst case in time complexity analysis. Therefore, according to the complexity of each step, the time complexity of the RBS algorithm is $O(W) + O(W) + O(W)$. In fact, the complexity of RBS algorithm is $O(W)$.

6 Experimental Evaluation

6.1 Simulation Setup

A discrete time simulation framework has been constructed in C++ language to experimentally evaluate the performance of the proposed algorithm. Runtime measurements have been carried out on a 1 GHz Pentium III computer. The simulated device consists of $96 \times 64 = 6144$ reconfigurable units (RCUs), which corresponds to Xilinx's XCV1000 FPGA.

We have conducted our experiments on four groups of input tasks. Each task group includes 20 task sets. Each task set includes 50 synthetic tasks. All synthetic tasks have randomly generated parameters. The probability distribution of task parameters are based on real tasks in real world applications (see Table 1)^[16]. The simulation results have been averaged for all task sets in each task group.

Table 1. Area Requirements for Typical FPGA Cores

Core	Area (RCUs)
UART ^[26]	50
100-Tap FIR Filter: 12-Bit Coefficients ^[27]	250
ADPCM ^[28]	250
DCT ^[29]	600
Triple-DES Processor ^[27]	800
256-Point Complex FFT ^[27]	850
Minimal Protocol Stack:	1 050
Ethernet MAC, IP, UDP ^[26]	
Discrete Wavelet Transform ^[29]	1 800

A parameter, called *repetition rate (RR)*, is defined for each task set as follows:

$$RR = \frac{\text{number of repetitive tasks in a task set}}{\text{total number of tasks in a task set}} \times 100\%. \quad (7)$$

In fact, parameter RR indicates the percentage of repetitive tasks in each task set. The characteristic of the generated task groups are as follows:

1) Task group A : $5 \leq RR_A \leq 15$. It means that repetition rate of all task sets in task group A is between 5 and 15.

2) Task group B : $15 < RR_B \leq 30$.

3) Task group C : $30 < RR_C \leq 60$.

4) Task group D : $RR_D = 0$.

Other parameters of the generated tasks in different task groups are the same as Table 2.

We use a workload parameter instead of the arrival time distribution in the figures. We denote the set of tasks arriving in the time interval ΔT with T . ΔT is set to span from the arrival of the first task to the arrival

Table 2. Distribution of the Task Parameters for All Task Groups (A, B, C, D)

Task Parameter	Width (w_i)	Execution Time (e_i)	Slack (S_i)	Reconfiguration to Execution Ratio (r_i/e_i)
Distribution	Uniform distribution in interval $[2, 40]$	Uniform distribution in interval $[5, 100]$	Uniform distribution in interval $[1, 50]$	Uniform distribution in interval $[0.1, 0.5]$

of the last task plus its execution time. Every task $T_i \in T$ occupies $w_i \times h_i$ reconfigurable units on the device area but as our area model is 1D, we use only w_i in workload definition. W is the total width of the RPU.

$$Workload = WL_{T,\Delta T} = \frac{\sum_{T_i \in T} w_i \times e_i}{W \times \Delta T}. \quad (8)$$

6.2 Simulation Results

We have compared the results of simulations for four different algorithms: our proposed Reusing-Based Scheduling (RBS) algorithm, Window-Based Stuffing (WBS) algorithm^[12] which is the improved version of stuffing algorithm^[16], Fragmentation-Aware algorithm presented by Handa (FA-H)^[24] and Fragmentation-Aware algorithm presented by Cui (FA-C)^[21]. All algorithms have been implemented for 1D area model. The obtained results from our implementations for WBS, FA-H and FA-C methods are close to the results obtained by their authors in similar situation. Therefore, the comparison results are reliable.

Figs. 6~9 indicate the task rejection ratio (see Subsection 2.2) of different algorithms for different task groups and Fig.10 compares the overall execution time of the tasks for task groups A, B, C and D using different scheduling algorithms. In these figures, LPR method has been used as replacement policy (see Subsection 4.4). As seen, when the repetition ratio is greater than 15%, task rejection ratio of the proposed algorithm (RBS) is close to the rejection ratio of FR-H and FR-C algorithms, while the overall execution time obtained by RBS is much better than the other algorithms. This is caused by high configuration reusing in RBS algorithm. Also, as expected, when the repetition

ratio of the input task group is equal to zero (task group D), the task rejection ratio and total execution time of the tasks for RBS method is not better than the other methods. In fact, in this situation, there is not superiority for the RBS method.

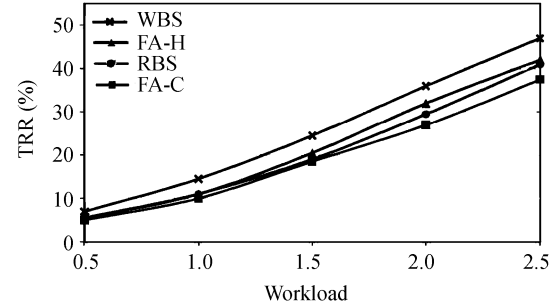


Fig.7. Comparison of task rejection ratio for different scheduling algorithms with task group B .

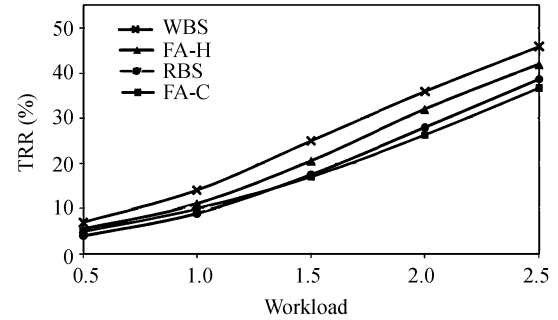


Fig.8. Comparison of task rejection ratio for different scheduling algorithms with task group C .

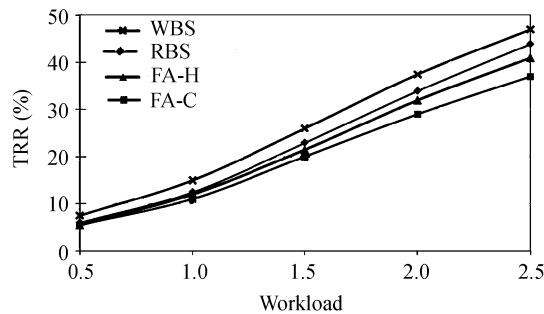


Fig.6. Comparison of task rejection ratio for different scheduling algorithms with task group A .

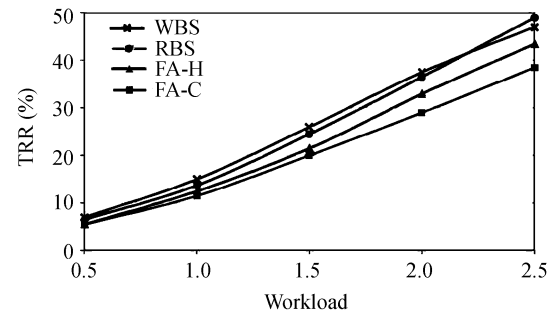


Fig.9. Comparison of task rejection ratio for different scheduling algorithms with task group D .

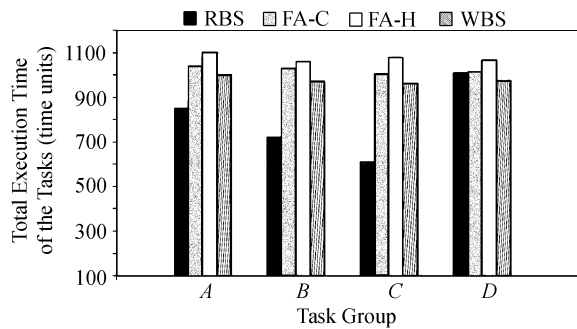


Fig.10. Comparison of total execution time of the tasks for different task groups using different scheduling algorithms.

Fig.11 compares the effect of using *best-fit* and *LPR* replacement policies in overall execution time of the tasks. As observed, *LPR* results in better execution time.

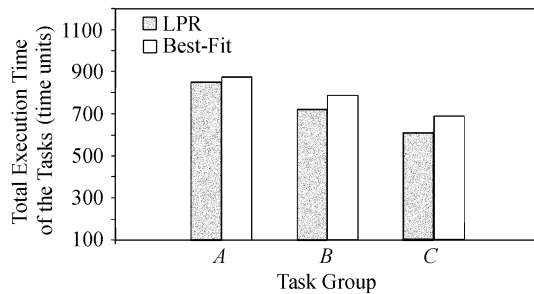


Fig.11. Impact of using different replacement policies on overall execution time in RBS scheduling algorithm.

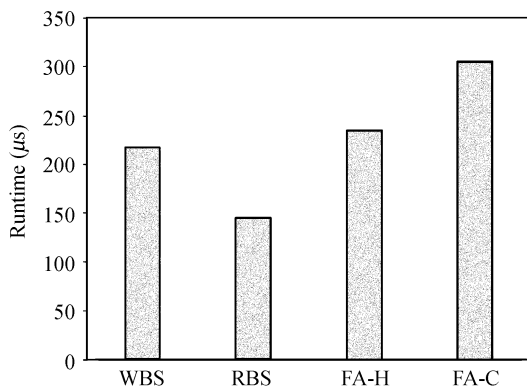


Fig.12. Runtime comparison of different scheduling algorithms.

Fig.12 shows the runtime comparison of different scheduling algorithms on 1 GHz Pentium III computer. As seen, the runtime complexity of the proposed algorithm is very small in comparison with two other algorithms, FA-H and FA-C.

7 Conclusion

In this work, we described the problem of on-line scheduling for RC systems with real-time,

non-preemptive task model and 1D area model.

We reviewed the related work and presented our proposed algorithm (RBS) which performs on-line scheduling and placement based on maximal configuration reusing (task reusing). The RBS method, partitions RPU surface into two virtual partitions and schedules arriving tasks on different partitions according to their significance. The significant tasks are maintained on the RPU after their completion as far as possible. Therefore it is possible to reuse the existing configurations when they recur in the future. The aim of proposed algorithm is to reduce the reconfiguration overhead of the tasks in order to improve the overall execution time of the application. Also, the proposed algorithm results in low task rejection ratio. Simulation results show these benefits of the proposed algorithm.

Acknowledgements Authors would like to thank from ITRC for supporting this research.

References

- [1] Saha P, El-Ghazawi T. A methodology for automating co-scheduling for reconfigurable computing systems. In *Proc. the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, Nice, France, May 30-Jun. 1, 2007, pp.159-168.
- [2] Walder H, Platzner M. Reconfigurable hardware operating systems: From concepts to realizations. In *Proc. the 3rd International Conf. Engineering of Reconfigurable Systems and Architectures (ERSA)*, Las Vegas, USA, Jun. 23-26, 2003, pp.284-287.
- [3] Deng Q, Wei S, Xu H, Han Y, Yu G. A reconfigurable RTOS with HW/SW co-scheduling for SOPC. In *Proc. the Second International Conference on Embedded Software and Systems (ICESS 2005)*, Xi'an, China, Dec. 16-18, 2005, pp.1-6.
- [4] Bazargan K, Kastner R, Sarrafzadeh M. Fast template placement for reconfigurable computing systems. In *IEEE Design and Test of Computers*, 2000, 17(1): 68-83.
- [5] Diessel O, ElGindy H, Middendorf M, Schmeck H, Schmidt B. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEEE Proceedings on Computers and Digital Techniques*, May 2000, 147(3): 181-188.
- [6] Marescaux T, Bartic A, Dideriek V, Vernalde S, Lauwereins R. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In *Proc. the 12th International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier, France, Sept. 2-4, 2002, pp.795-805.
- [7] Virtex 2.5 V Field Programmable Gate Arrays. Xilinx, Inc. December 2002.
- [8] Banerjee S, Bozorgzadeh E, Dutt N. Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration. *IEEE Transaction on Very Large Scale Integration (VLSI)*, November 2006, 14(11): 1189-1209.
- [9] Noguera J, Badia R M. A HW/SW partitioning algorithm for dynamically reconfigurable architectures. In *Proc. DATE*, Munich, Germany, Mar. 12-16, 2001, pp.729-734.
- [10] Li Z, Hauck S. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proc. FPGA 2002*, Monterey, USA, Feb. 24-26, 2002, pp.187-195.
- [11] Huang M, Simmler H, Saha P, El-Ghazawi T. Hardware task scheduling optimizations for reconfigurable computing. In

- Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Austin, USA, Nov. 17, 2008, pp.1-10.
- [12] Zhou X G, Wang Y, Huang X Z, Peng C L. On-line scheduling of real-time tasks for reconfigurable computing system. In *Proc. FPT Conference*, Bangkok, Thailand, Dec. 13-15, 2006, pp.57-64.
- [13] Danne K, Platzner M. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In *Proc. International Conference on Field Programmable Logic and Applications*, Tampere, Finland, Aug. 24-26, 2005, pp.568-573.
- [14] Ahmadiania A, Bobda C, Teich J. A dynamic scheduling and placement algorithm for reconfigurable hardware. In *Proc. ARCS*, Augsburg, Germany, Mar. 23-26, 2004, pp.125-139.
- [15] Walder H, Platzner M. Non-preemptive multitasking on FPGA: Task placement and footprint transform. In *Proc. the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, Jun. 2002, pp.24-30.
- [16] Steiger C, Walder H, Platzner M. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *Proc. the 13th International Conference on Field Programmable Logic and Application (FPL 2003)*, Lisbon, Portugal, Sept. 1-3, 2003, pp.575-584.
- [17] Marconi T, Lu Y, Bertels K, Gaydadjiev G. Online hardware task scheduling and placement algorithm on partially reconfigurable devices. In *Proc. ARCS*, Dresden, Germany, Feb. 25-28, 2008, pp.306-311.
- [18] Steiger C, Walder H, Platmer M, Thiele L. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *Proc. RTSS 2003*, Cancun, Mexico, Dec. 3-5, 2003, pp.224-235.
- [19] Qiu W D, Zhou B, Chen Y, Peng C L. Fast on-line real-time scheduling algorithm for reconfigurable computing. In *Proc. the Ninth International Conference on Computer Supported Cooperative Work in Design*, Coventry, UK, May 24-26, 2005, pp.793-798.
- [20] Roman S, Mecha H, Mozos D, Septien J. Constant complexity scheduling for hardware multitasking in two dimensional reconfigurable field-programmable gate arrays. *Journal of IET Comput. Digit. Tech.*, 2008, 2(6): 401-412.
- [21] Cui J, Gu Z, Liu W, Deng Q. An efficient algorithm for on-line soft real-time task placement on reconfigurable hardware devices. In *Proc. the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, Santorini Island, Greece, May 7-9, 2007, pp.321-328.
- [22] Dittmann F, Frank S. Caching in real-time reconfiguration port scheduling. In *Proc. FPL 2007*, Amsterdam, The Netherlands, Aug. 27-29, 2007, pp.740-744.
- [23] Liang L, Zhou X, Wang Y, Peng C. On-line hybrid task scheduling in reconfigurable systems. In *Proc. the 11th International Conference on Computer Supported Cooperative Work in Design*, Melbourne, Australia, Apr. 26-28, 2007, pp.1072-1077.
- [24] Handa M, Vemuri R. Area fragmentation in reconfigurable operating systems. In *Proc. ERSa*, 2004, pp.77-83.
- [25] Cui J, Deng Q, He X, Gu Z. An efficient algorithm for online management of 2D area of partially reconfigurable FPGAs. In *Proc. Design, Automation and Test in Europe (DATE)*, Nice, France, Apr. 16-20, 2007, pp.129-134.
- [26] Lerjen M, Zbinden C. Reconfigurable bluetooth-Ethernet bridge [Master's Thesis]. Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, 2002.
- [27] Xilinx Inc. Virtex Core Generator.
- [28] Dyer M, Wirz M. Reconfigurable system on FPGA [Master's Thesis]. Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, March 2002.
- [29] Amphion Semiconductor Ltd. www.amphion.com.



Maisam M. Bassiri received the B.S. and M.S. degrees in electrical engineering from Sharif University of Technology (Iran) and Amirkabir University of Technology (Iran) in 2002 and 2004, respectively. Now he is Ph.D. candidate in electrical engineering in Iran University of Science and Technology. His research interests are in the field of parallel processing, embedded system design and reconfigurable computing. Now, his current research focuses on HW/SW co-design and resource management in reconfigurable computing systems. Also he has published several papers in his research fields so far.



Hadi Shahriar Shahhoseini received the B.S. degree in electrical engineering from University of Tehran, in 1990, the M.S. degree in electrical engineering from Azad University of Tehran in 1994, and the Ph.D. degree in electrical engineering from Iran University of Science and Technology, in 1999. He is an assistant professor of the Electrical Engineering Department in Iran University of Science and Technology. His areas of research include networking, supercomputing and reconfigurable computing. More than 130 papers have been published from his research works in scientific journals and conference proceedings. He is an executive committee member of IEEE TCSC and serves IEEE TCSC as regional coordinator in middle-east countries.