社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2006－ＳＬＤＭ－125（１）
2006／5／11

I/Oルーティング情報を用いたオンラインFPGAプレイスメント

伴野 充† 中西 正樹† 山下 茂† 中島 和生†† 渡邉 勝正†

† 奈良先端科学技術大学院大学
†† メリーランド大学，カレッジパーク校
E-mail: †{mitsu-to,m-naka,ger,watanabe}@is.naist.jp, ††nakajima@umd.edu

あらまし 部分再構成可能FPGAでは，ロジックリソースとインターコネクション網の任意の部分を動的に再構成することが可能となる．そのため，FPGA上にて同時に多数のタスクを実行することが出来る．限られたFPGAリソースを用いて効率的にタスクを実行するためには，FPGAリソースの管理が極めて重要となる．近年，いくつかのオンラインFPGAプレイスメント手法が提案されたが，そのどれもがタスクのI/O通信を扱うことが出来ない．タスクのI/O通信を明示的に扱うことで，我々は新しいオンラインプレイスメント手法を提案する．我々の手法は各空き領域に付随するI/Oエレメントまでの経路情報を用いることで，各タスクに対して適切な領域を選択する．

キーワード オンラインプレイスメント，部分再構成可能FPGA，リコンフィギャラブルコンピューティング

Online FPGA Placement Using I/O Routing Information

Mitsuru TOMONO†, Masaki NAKANISHI†, Shigeru YAMASHITA†, Kazuo NAKAJIMA††, and Katsumasa WATANABE†

† Nara Institute of Science and Technology
†† University of Maryland, College Park
E-mail: †{mitsu-to,m-naka,ger,watanabe}@is.naist.jp, ††nakajima@umd.edu

Abstract In a partially reconfigurable FPGA, arbitrary portions of its logic resources and interconnection networks can be reconfigured without affecting the other parts. Thus, several tasks can be mapped and executed concurrently in the FPGA. In order to execute the tasks efficiently using the limited resources of the FPGA, resource management becomes very important. Although some online FPGA placement methods have, recently, been proposed, they cannot handle I/O communications of the tasks. Taking such I/O communications into consideration, we introduce a new approach to online FPGA placement. We present a task placement algorithm which uses I/O routing information of each empty area to select a suitable area for each task. The algorithm uses a combination of two new fitting strategies and modified versions of two existing strategies.

Key words Online Placement, Partially Reconfigurable FPGAs, Reconfigurable Computing

1. Introduction

The rapid technological advancement of FPGAs (Field Programmable Gate Arrays) has recently resulted in the production of partially reconfigurable FPGAs on the market. In such devices, arbitrary portions of their logic resources and interconnection networks can dynamically be reconfigured without affecting the other parts of the system to be implemented. This allows multiple tasks to be executed in parallel by hardware. For such systems an application is first divided into a set of small tasks, called hardware tasks. Each of them is then placed in an area of a sufficient size when the system requests its execution. When the task is completed, the system deletes it and its assigned area is reformed and reused by other tasks.

The partial reconfigurability may increase the FPGA resource utilization. On the other hand, the FPGA surface would be fragmented [3], unless proper management of the device resources is provided. As a result, the area utilization of the FPGA may decrease and a newly arrived task may not be placed although many empty areas exist. This necessitates the development of an efficient FPGA resource management method. Such placement management is provided by a so-called *reconfigurable operating system* (ROS) [3]. In particular, a special module, called task placement module, of the ROS provides a set of system services such as task scheduling, task loading, and task addition and deletion.

The task placement on FPGAs may be classified into two categories: offline and online. In the offline placement, the flow of tasks is known in advance. Various optimization algorithms such as simulated annealing and genetic algorithms

have been applied to obtain good quality placements [2].

On the other hand, in the online placement, the sequence of task requests is known only at run time. Thus the system needs to handle each task on a case-by-case basis. If the time needed for the placement increases, it becomes overhead to the system. Therefore, it becomes more important to balance the time for and the quality of placement.

To our best knowledge, Bazargan, et al. [1] was the first to introduce an approach to online task placement. They proposed a rectangular task model, developed several online placement algorithms, and did simulations with a few fitting strategies. Their empty area partitioning algorithm was a heuristic and did not produce optimal solutions. Walder, et al. [5] presented an improved version of this partitioning algorithm and a method of locating feasible empty areas for a task. They introduced an array called *Hash Matrix* to store pointers to lists of empty areas of different sizes.

Handa and Vemuri [4] developed several algorithms for online placement and task scheduling. They took a computational geometry based approach and used algorithms for finding and maintaining empty areas on an FPGA surface in the development of their online placement algorithm. Their method maintains such empty areas as so-called *maximal empty rectangles* (MER) [1], which may overlap with each other [4].

All of the algorithms mentioned above are based on the rectangular task placement model [1]. However, their placement model ignored communications between the tasks and the I/O elements located on the periphery of the FPGA. In this paper, we introduce a new model for online task placement on partially reconfigurable FPGAs. Although it is based on the rectangular task model, our model considers I/O communications of the tasks. We present an online task placement algorithm which uses a combination of four fitting strategies to find a most preferable empty area for each task. Two of the strategies are new and explicitly reflect on I/O routing information of each empty area.

In the next section, we present our model for a reconfigurable computing system. In Section 3., we explain the two new fitting strategies and our online task placement algorithm. We then present a set of evaluation criteria and show our simulation results in Section 4.. Section 5. summerizes the paper and describes some future work.

2. Model of a Reconfigurable Computing System

We describe our model for a reconfigurable computing system in detail. Each aspect of the model is presented in a separate subsection.

2.1 System Model

Figure 1 depicts our system model consisting mainly of a host CPU, a shared memory, a ROS, and a partially reconfigurable FPGA. The ROS runs on the host CPU and has the placement module that manages the FPGA resources. It is composed of a scheduler, a loader, and a placer, which provide a system service of task scheduling, task loading, and task addition and deletion, respectively. The shared memory is used to provide configuration data of each task and stores the results of task execution.

We show the flow of processing of the online FPGA placement in Figure 2. The ROS requests a task execution to its placement module when a new task arrives. The scheduler receives the request and places it in its queue. It then finds a task in its queue to be placed. In the online placement, scheduling is done basically in the order of arrivals of the tasks, or, at best, by exploiting their priorities in some measures. After that, the placer searches the list of empty

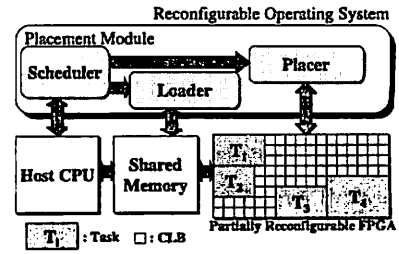

Figure 1 System Model

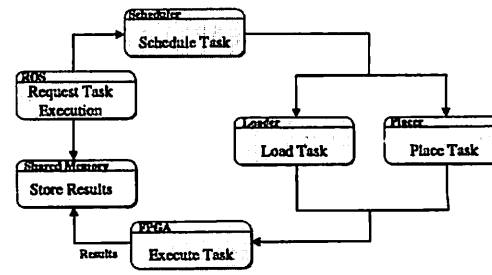

Figure 2 Flow of Processing

rectangular areas for the task to be placed next. If an area large enough for the task is found, the loader loads its configuration data into the FPGA. Then, the loaded task starts its execution. At its completion, the loader writes the results into the shared memory.

2.2 FPGA Model

Our FPGA consists of CLBs uniformly arranged in the form of a two dimensional array. Note that FPGA products from Xilinx such as its Vertex series [6] use CLBs as basic logic units. In our FPGA model, we assume that each CLB can be reconfigured independently. In other words, each CLB can change its configuration at run time without affecting the other CLBs. This ability of the FPGA is called *Partial Reconfigurability*.

Communication channels are provided along every side of each CLB. Each CLB can freely access to its neighboring communication channels. So any pair of tasks can communicate with each other in theory. However, in our model, we assume that communications take place between a task and an I/O element located on the periphery of the FPGA. The I/O elements then manage communications to external components such as the host CPU and the shared memory. Our FPGA model further assumes the existence of unlimited numbers of communication channels so that communications between any pair of a task and an I/O element may be realized.

In our model, we consider communication costs. A unit time delay of communication is associated with the single step traversal of a signal, where a single step is measured as the length of a channel along a single side of each square CLB.

2.3 Task Model

We now describe our assumptions on some features of a task in our model. The shape of a task is rectangular.

We also assume that different clocks may propagate to each CLB. Thus, each task may operate at a different clock frequency, and may also require multiple cycles to execute the function of the task. Therefore, the actual execution time of each task is obtained by (its required number of cycles) ×

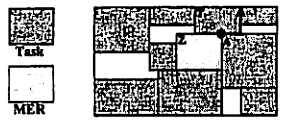

Figure 3 Task Boundary = Route

(its clock frequency).

Each task communicates its data through an I/O element asynchronously. Thus, it has the amount of input and output data communication bits.

The I/O port of each task is placed at a corner of its rectangle. Empty areas of the FPGA are also managed in a rectangular shape. Each empty area has routing information to an I/O element. Such information includes the number of steps and channel information to the I/O element. A port to the I/O element of each empty area is placed at one of the four corners of the rectangle. Therefore, to fit the I/O port of the task into that of an empty area, four types of hardware macros are prepared for each task. Each type of the macros has the I/O port at a different corner of a rectangle.

The parameters of each task are summarized as follows.

- Task Width
- Task Height
- The Number of CLBs
- Required Cycles
- Clock Frequency
- Data Communication Bits

2.4 Problem Modeling

In the online FPGA placement, the most important issue is the way in which the placer of the placement module handles newly arrived tasks. The placer receives a sequence of tasks from the scheduler as inputs. Outputs of the placer are feasible empty areas for those upcoming tasks. Thus, the objective of our placer is to place tasks online in the FPGA in such a way that the total execution time becomes shorter.

2.5 Constraints on I/O Communications

In order to make an I/O communication path from an empty area of the FPGA to an I/O element, we use the boundaries of the tasks as much as possible. In Figure 3, dark and light gray rectangles represent, respectively, tasks and empty areas. For illustrative purposes, only the intended empty areas are shown in this and the following figures. Figure 3 depicts a case in which task boundaries constitute a communication path. In this figure, the empty area designated by Z has a path from vertex A to the I/O element through the boundary of Task T. The communication path of the empty area Z does not affect other empty areas since it passes through task boundaries only.

On the other hands, Figure 4 shows a case in which no task boundary is part of a communication path. The figure illustrates the task placement right after Task T finishes and is deleted from the FPGA. There are three overlapping empty areas, X, Y and Z in the area where Task T was placed and the empty area Z has a straight path from vertex A to the upper edge of the FPGA through the empty area X. As a result, the empty area X is split into two empty areas L and M, and the empty area L is then merged with the empty area Y. In this way we have lost the empty area X due to the path of the empty area Z through the empty area X. Therefore, our model uses task boundaries for a communication path of each empty area as much as possible.

2.6 Evaluation Methods

We present two methods of evaluation of an FPGA task placement, depending on whether the tasks are independent or dependent. An execution process of a task is depicted in

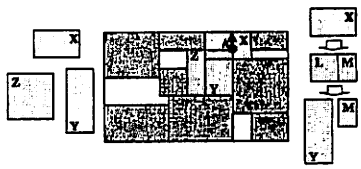

Figure 4 Task Boundary ≠ Route

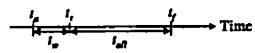

Figure 5 Execution Process of a Task

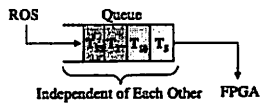

Figure 6 Independent Tasks

Figure 5, where the parameters denote the following.

t_a : *arrival time*
t_s : *start time*
t_f : *finish time*
t_w : *waiting time*
t_e : *execution time*
t_c : *communication time*
t_{all} : $t_e + t_c$

Independent Task Environment

When the tasks come from different applications, their executions do not depend on each other. Figure 6 shows an example. We define the term *average waiting time* for n independent tasks as

$$\overline{t_w} = \frac{1}{n}\sum_{i=1}^{n} t_{w_i} \tag{1}$$

The value of $\overline{t_w}$ measures how quickly the tasks are accepted by the placement algorithm. In order to measure the efficiency of task communications including waiting time, we define the term *average overhead time* as

$$\overline{t_{oh}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{t_{c_i} + t_{w_i}}{t_{c_i}}\right) \tag{2}$$

For n independent tasks, if $\overline{t_w}$ is small, it means that each task can start its execution without waiting for a long time. If $\overline{t_{oh}}$ is small, the processing of tasks is done efficiently in terms of task communications.

Dependent Task Environment

When the tasks come from the same application, some tasks may not be able to start their execution until other tasks finish. For n such dependent tasks in the same application, $max(t_{f_j})$ represents the finish time at which the last task finishes, and $min(t_{a_i})$ the arrival time at which the first task arrives. The total execution time of the application is given by

$$t_{total} = \max_j(t_{f_j}) - \min_i(t_{a_i}) \tag{3}$$

Figure 7 shows an example of the data flow of an application. In the figure, tasks T_1 and T_9 are the first and the last task, respectively, in the application. The difference between the arrival time of task T_1 and the finish time of T_9 is the total execution time of the application. Therefore, the smaller t_{total} is, the faster the system could execute an application.

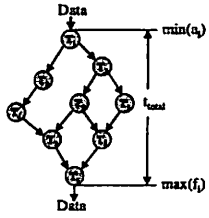

Figure 7 Dependent Tasks

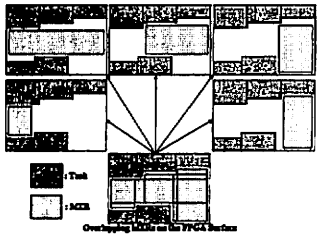

Figure 8 Overlapping Maximal Empty Rectangles

3. Task Placement Algorithm

We start with an overview of our task placement method that handles I/O communications of the tasks. The main modules of our placement engine are a scheduler, a loader, and a placer. Among them, the placer plays a major role. Thus we focus on the four components of the placer: an FPGA surface manager, an I/O routing engine, an empty area manager, and a fitter. The surface manager provides a way to obtain empty areas on the FPGA surface and is described in Section 3.1. The routing engine provides an algorithm for creation of routing information for the empty areas obtained and is presented in Section 3.2. The empty area manager gives a method of management of the empty areas and is explained in Section 3.3. The fitter selects a suitable area from a group of empty areas by use of fitting strategies that are given in Section 3.4.

We briefly describe the relationship among these components. At the time of task addition and deletion, the FPGA surface manager updates a data structure that represents a state of the FPGA surface. When it extracts an empty area from the data structure, the surface manager sends the extracted empty areas to the empty area manager for storage. The routing engine creates a communication path for each stored empty area. When a new task arrives, the fitter examines the empty areas and finds a suitable area for the task based on the fitting strategies.

3.1 Management of FPGA Surface

We present the first component of our task placer. We need an efficient management of the FPGA surface for full utilization of partially reconfigurable FPGAs. For this, we use a modified version of the method proposed by Handa, et al. [4].

We start with the following definition [4].

[Definition 1] A *maximal empty rectangle* (MER) is the empty rectangle that can not be fully covered by any other empty rectangle. □

We use MERs to manage the empty areas on an FPGA surface. Examples of MERs are depicted in Figure 8, where dark and light gray rectangles represent tasks and MERs, respectively, and some of the MERs overlap with each other.

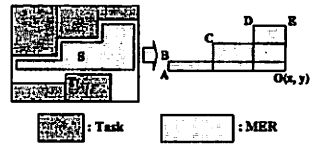

Figure 9 Staircase

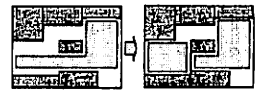

Figure 10 Partitioning of Staircases

We also need the following definitions.

[Definition 2] A *staircase* (x, y) is defined as a collection of all empty rectangles with point (x, y) as their lowest right vertex. The point (x, y) is called the *origin* of the staircase. □

We use a data structure, called area matrix, to construct MERs. We scan rows of the area matrix and make a staircase as shown in Figure 9. Since each cell P of the area matrix keeps the number of empty cells at and above P in its column, we can construct a staircase by scanning rows. For example, in Figure 9, we can obtain staircase S with the origin O by scanning the row just above the top horizontal boundary of task T. In this way, MERs OB, OC,and OD are obtained. It has been proven that each staircase always rests on the top horizontal boundary of an already placed task [4]. Therefore, to update the state of the FPGA, it is sufficient for the placer to scan only the rows just above the top horizontal boundaries of the tasks already placed.

In our model, the existence of an I/O communication path for a task may necessitate the partitioning of a staircase into two as depicted in the right half of Figure 10. In this case, we need to construct a new staircase from the I/O communication path.

3.2 I/O Routing Algorithm

The second component of our task placer is an I/O routing engine. We use a greedy algorithm of construction of a communication path from a task to an I/O element located on the periphery of an FPGA. By using this component, the length of the path is also determined.

3.3 Management of MERs

We now turn our attention to the third component, an empty area manager, which also plays an important role in the delivery of good task placement. We start with the following definition of the data structure introduced by Walder, et al. [5].

[Definition 3] A *Hash Matrix* is an array of $h \times w$, where the parameters h and w denote the numbers of rows and columns, respectively, of the CLBs in an FPGA. $Entry[a][b]$ of a *Hash Matrix* is associated with MERs of size $a \times b$. It has two pointers. One is a pointer to a list of all the MERs of size $a \times b$. The other is a pointer to a list of MERs of size $a' \times b'$ such that $a' \geq a$, $b' \geq b$, $a' \times b' > a \times b$, and $a' \times b'$ is closest in value to $a \times b$. □

When it attempts to place a task of size $a \times b$, the system may need to examine all rectangles of size greater than or equal to $a \times b$, depending on the strategies used in the fitter that is to be explained in the next section. In this case, the MER manager checks out only at the entries $Entry[a'][b']$ with $a' \geq a$ and $b' \geq b$ in the *Hash Matrix*.

3.4 Fitting Strategies and Cost Functions

We finally describe the fourth component, the fitter, of our

task placer. When multiple MER candidates for a newly arrived task are available, the selection of a particular MER is determined by a combination of four strategies: (1) Best Fit, (2) I/O Oriented, (3) Route Duration, and (4) Fragmentation Aware. The first strategy, *Best Fit* has been used as the main strategy in the conventional model, while the fourth one, *Fragmentation Aware* was proposed by Handa et al. [3]. We introduce strategies (2) *I/O Oriented* and (3) *Path Duration* to deal with I/O communications of tasks in our model. For each strategy we define its associated cost function.

Strategy 1: *Best Fit*

Under this strategy we select, from the pool of MERs, a smallest MER that can accommodate the arrived task. Let S_{MER} and S_{task} denote the numbers of CLBs that an MER contains and the task needs, respectively. The cost function for Best Fit is given as follows.

$$Cost_{BF} = \frac{S_{MER}}{S_{task}} \tag{4}$$

The smaller $Cost_{BF}$ is, the smaller number of CLBs of the MER is wasted.

Strategy 2: *I/O Oriented*

Under this strategy, we select an MER that can provide faster communication of data for the arrived task. Each task has a parameter C_{bit} that indicates the amount of its communication bits. Each MER holds a parameter D_{route} which denotes the number of steps of its communication path to an I/O element. Let U_{time} be the delay time for a task to route its data through a 1 step communication channel. Let B_{width} denote the number of communication channels available per task. The cost function for the I/O Oriented strategy is now defined.

$$Cost_{IO} = D_{route} \times U_{time} \frac{C_{bit}}{B_{width}} \tag{5}$$

Strategy 3: *Path Duration*

Suppose that some of the communication channels between two neighboring tasks are used as the I/O communication path for data of a third task. When the two neighboring tasks are completed, two fragmented MERs may reside, rather than a combined one, if the channels are still used for the third task. In order to avoid this type of fragmentation caused by communication paths as much as possible, we need a new fitting strategy in our model. We define the duration of path of an MER as the average over the durations of all tasks that are border on the path of the MER. Under the new strategy, we select an MER whose duration of path is as close to the duration of the arrived task as possible.

An I/O communication path of an MER passes along task boundaries. So we define a parameter t_{path} as the average over the finish times of the tasks that are border on the path. Let $t_{current}$ denote the system time when the task arrived. Note that t_f is the finish time of the arrived task. We define the cost function for Path Duration as follows.

$$Cost_{PD} = \begin{cases} 0 & (t_f \leqq t_{path}) \\ \frac{t_f - t_{current}}{t_{pass} - t_{current}} & (t_f > t_{path}) \end{cases} \tag{6}$$

If $t_f \leqq t_{path}$, the newly arrived task and its communication path disappear before all of the tasks border on the path are completed and hence $Cost_{PD}$ is set to 0. If $t_f > t_{path}$, however, some of the tasks that are border on the path of an MER finish earlier than the arrived task. As a result, this path still remains on the FPGA surface and may create fragmentation of the surface. Thus, it is preferable to select an MER whose t_{path} is closest to t_f of the arrived task.

Strategy 4: *Fragmentation Aware*

Under this strategy we try to select an MER from areas with more tasks allocated so as to prevent other less crowded areas from being fragmented. Handa, et al. [3] introduced a parameter to quantify the fragmentation of an FPGA surface and provided detailed discussions on this strategy. For completeness, we simply give their definition of the cost function for Fragmentation Aware. Let $TFCC$ and TF, respectively, denote what they call Total Fragmentation Contribution of a Cell or a CLB and Total Fragmentation for an MER. A larger TF means that its MER is more fragmented.

$$TF = \frac{\sum_C TFCC}{S_{MER}} \times 100 \tag{7}$$

$$Cost_{FA} = \frac{1}{TF} \tag{8}$$

In order to reduce the fragmentation of the FPGA surface, the placer selects an MER with a smaller $Cost_{FA}$.

We now define the total cost function as a weighted sum of the above four cost functions. Let α, β, γ, and δ be user-defined parameters. By proper selections of values for these parameters, the placer will be able to place different levels of emphasis on the strategies.

$$Cost_{ALL} = \alpha \cdot Cost_{BF} + \beta \cdot Cost_{IO} + \gamma \cdot Cost_{PD} + \delta \cdot Cost_{FA} \tag{9}$$

When a task arrives, the placer calculates this value for all MERs of feasible sizes. It then selects an MER with minimum $Cost_{ALL}$ for the task.

4. Evaluation of Our Placement Engine

In order to evaluate the effectiveness of the four fitting strategies described above and their combinations, we conducted simulations for the case of an FPGA with 96 × 64 CLBs. Ten sets of 500 tasks each are randomly generated for each experimental environment and the results to be shown below are the average over these 10 sets. By setting appropriate parameter values, three different task sets, called a small, medium, and large task set, are created. Communication bits for the tasks are randomly produced between 1 and 128. The execution time of each task and intervals between two consecutive task arrivals are also randomly generated. The values of w_{band} and t_{unit} are set to 8 and 1, respectively.

Most of the previously proposed algorithms in the conventional model mainly used Strategy 1, *Best Fit*. Thus, we assume the case in which only Strategy 1 is used, namely, $\alpha = 1$ and $\beta = \gamma = \delta = 0$ corresponds to the conventional method. We designate this case as Case 1. Likewise, the case in which a single strategy i is used, is called Case i for $i = 2, 3, 4$. More precisely, the parameter values for each case are set as follows:

Case 2:$\alpha = 0,\ \beta = 1,\ \gamma = \delta = 0$
Case 3:$\alpha = \beta = 0,\ \gamma = 1,\ \delta = 0$
Case 4:$\alpha = \beta = \gamma = 0,\ \delta = 1$

In each experiment with 10 sets of tasks, we ran our online placement algorithm with certain strategies emphasized over the others. We measured the total execution time. In order to observe the performance of our new fitting strategies proposed for our model with I/O communications, we divided the values obtained in these three measures by their corresponding values for Case 1 (i.e., the conventional method). In each figure to follow, we show these fractional values for these four cases. Note that those values for Case 1 are always 100%.

After having observed the results for each of Cases 2, 3, and 4 as compared with those for Case 1 for the task sets of three different sizes, we carefully selected the values for parameters α, β, γ, and δ. In particular, the simulation results

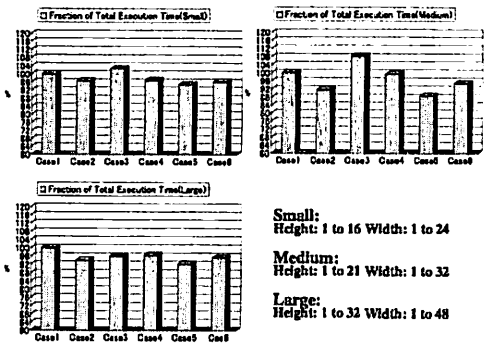

Figure 11 Performance Summary for Three Task Sets

indicated a good performance improvement by Strategy 2 of *I/O Oriented* for each of the task set categories. We designate the case of these new parameter values for the total cost function as Case 5.

For each of these three different task sets, we again ran the placement algorithm and obtained the three values. The fractions of these measured values over those of Case 1 are provided as the Case 5 data in each figure.

In addition, we consider a default case in which the parameters are set as $\alpha = 1$, $\beta = 5$, $\gamma = 1$ and $\delta = 5$. As noted earlier, the *I/O Oriented* strategy always plays an important role. Likewise, as noted later, the *Fragmentation Aware* strategy has an impact, in particular, on the small task set. When the characteristics of tasks are not known in advance, this parameter setting enters into play. We denote this case as Case 6. For completeness of the presentation of our experimental results, we add the results for Case 6 to each figure.

We are now ready to present our experimental results for each task set category. Note that the designation of each category, small, medium, and large, means that the largest task in each set is small, medium, and large, respectively, in size. The tasks in a small task set are of height between 1 and 16 and width between 1 and 24. Note that the largest values in task height and width, respectively, are a quarter of those for the FPGA. We show the experimental results in Figure 11. The data for Cases 2 and 4 clearly indicated performance improvements by their corresponding strategies, *I/O Oriented* and *Fragmentation Aware*. We therefore set the parameter values for Case 5 as $\alpha = 5$, $\beta = 40$, $\gamma = 1$, and $\delta = 30$.

In this small task set, area fragmentation would most likely to occur since there are a lot of small tasks placed in the FPGA. Therefore, we assigned a value of 30 to the parameter δ to incorporate a relatively high impact by the *Fragmentation Aware* fitting strategy. As mentioned earlier, the *I/O Oriented* strategy always improves the results in any task set category. So we set $\beta = 40$ for Case 5. It should be noted that this case further improved the results.

In the medium task set category, task heights and widths are between 1 and 21 and between 1 and 32, respectively. Note that the maximum height and width each are set to be a third of the FPGA height and width.

By comparing the results for Case 4 in this and the previous figures, we observe that *Fragmentation Aware* has less impact in the medium task set than in the small task set. This is because a greater number of larger tasks would likely to reduce area fragmentation. Note also that in both the small and medium task sets, the *Path Duration* strategy did not produce good results. This is most likely due to the facts that a lot of tasks are placed in the FPGA and that the finish times of the tasks vary very much. Therefore, we set the parameters of Case 5 as $\alpha = 1$, $\beta = 10$, $\gamma = 0$, and $\delta = 1$.

In the large task set, the tasks have heights between 1 and 32 and widths between 1 and 48. Note that the largest task height and width are set to a half of the FPGA height and width, respectively.

The figure shows that the *Path Duration* strategy now has a positive effect on the results. Since more relatively large tasks exist in the set, there are a smaller number of tasks being executed simultaneously in the FPGA, as compared to the small and medium task sets. Furthermore, the large task sizes would most likely not to produce many small fragmentations. Thus, the effect of the *Fragmentation Aware* strategy becomes low as indicated in the figure. Therefore, we set the values of parameters of Case 5 as $\alpha = 1$, $\beta = 100$, $\gamma = 10$, and $\delta = 5$.

5. Conclusions

In this paper, we introduced a new model for online FPGA placement. Unlike the conventional model, our model considers the effect of communications between the tasks and the I/O elements on the periphery of a partially reconfigurable FPGA. We proposed two fitting strategies for task placement in our model. We developed an online task placement algorithm, which selects an empty area for each task using a combination of these two and two other previously used fitting strategies. Together with two existing strategies, we showed by simulation the effectiveness of our strategies of our placement algorithm.

In the next step of our research, in our model we assume the existence of an unlimited amount of communication channels. In the near future, we will conduct research for the case of limited communication resources. In such a system, it would make sense to keep some empty areas between placed tasks so as to accommodate future needs of I/O communications of new arrivals. We plan to introduce a new FPGA model and develop a new online task placement method for such a model.

Acknowledgment

This work was partially supported by MEXT.KAKENHI ((C)(2)15500023) and ((B)16700067) and the Okawa Foundation Research Grant.

References

[1] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design and Test of Computers*, Vol. 17, pp. 68–83, 2000.

[2] S. Fekete, E. Köhler, and J. Teich, "Optimal FPGA Module Placement with Temporal Precedence Constraints," *Proc. of Design Automation and Test in Europe Conf. and Exhibition*, Munich, Germany, 2001, pp. 658–665.

[3] M. Handa and R. Vemuri, "Area Fragmentation in Reconfigurable Operating Systems," *Proc. of International Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, June, 2004.

[4] M. Handa and R. Vemuri, "An Efficient Algorithm for Finding Empty Space for Online FPGA Placement," *Proc. of 41st Design Automation Conf.*, San Diego, CA, June, 2004, pp. 960–965.

[5] H. Walder, C. Steiger, and M. Platzner, "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing,"

[6] Xilinx, Inc. http://www.xilinx.com/