NAIST-IS-DD0461021

Doctoral Dissertation

Studies on Utilization Schemes for Reconfigurable Computing Systems

Mitsuru Tomono

March 23, 2007

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Mitsuru Tomono

Thesis Committee:

Professor Yasuhiko Nakashima	(Supervisor)
Professor Hideo Fujiwara	(Co-supervisor)
Associate Professor Shigeru Yamashita	(Co-supervisor)

Studies on Utilization Schemes for Reconfigurable Computing Systems[*]

Mitsuru Tomono

Abstract

Reconfigurable computing has now become a promising way to improve performance of a computer system, whereby solution algorithms of target applications are mapped to reconfigurable components of a system. Overall system performance is significantly improved by processing critical parts of algorithms using hardware resources. Reconfigurable computing systems provide the benefits of the flexibility of general-purpose processors and the high performance of dedicated systems. They have shown their effectiveness in applications such as image processing, data encryption, and pattern matching. However, various problems such as their rudimentary application areas and insufficient design environment still limit their utilization.

In this dissertation, utilization schemes for reconfigurable computing systems are proposed. First, an exploitation method for a system that is composed of a general-purpose processor and reconfigurable logics is presented by applying Event-Oriented Computing (EOC). One characteristic of EOC is that it is suitable for such a reconfigurable hybrid system. An architecture model for EOC is proposed, with experimental results demonstrating its performance-improvement capabilities.

Second, a new approach to online task placement is presented. In partially reconfigurable Field Programmable Gate Arrays (FPGAs), multiple tasks can be executed in parallel by hardware. In such systems, effective FPGA resource management is necessary to process incoming tasks efficiently. The proposed

[*] Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0461021, March 23, 2007.

online task placement algorithm leverages I/O routing information of tasks for its placement process. The effectiveness of the placement engine is also shown in comparison with conventional methods.

Third, a placement and routing algorithm for a reconfigurable 1-bit processor array (1-bit RPA) is proposed. The 1-bit RPA employs a bit-serial data path and a unique wiring structure. The architectural difference between a 1-bit RPA and an FPGA is that the former does not allow a simple application of mapping techniques of FPGAs. The dedicated placement and routing algorithm for the 1-bit RPA is proposed. In the algorithm, empty processor elements are placed deliberately in the initial placement stage and used in the initial routing stage. The subsequent optimization stages utilize these empty cells to improve the placement quality.

Keywords:

algorithm, reconfigurable computing, reconfigurable hardware, FPGA, online task placement, placement and routing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Overview

In recent years, researches on reconfigurable computing has extended [7], [10], [42] and many computer architectures utilizing reconfigurable logics have been proposed [11], [25], [36]. They improve system performance by directly mapping solution algorithms to reconfigurable components of systems. The architectures have shown their effectiveness in some applications such as multimedia applications, data encryptions, and so on [2], [41]. Reconfigurable devices can also meet the needs of changes to the specification after fabrication by utilizing their flexibility.

Among many architectures proposed, Field Programmable Gate Arrays (FPGAs) [6] are widely used due to their flexibility, cost, and well-developed design environment [5]. Many recent digital appliances such as digital television or automotive navigation equipment employ FPGAs in their system. In particular, they establish a presence in prototyping or debugging of large scale integrated circuits (LSIs) since it is difficult for LSIs to change their functions and fix bugs after fabrication. The architectural features of FPGAs are a fine-grained structure and static reconfiguration. In addition to the FPGAs, dynamically reconfigurable coarse-grained architectures have also been researched extensively [3].

Reconfigurability of hardware aims to realize flexibility and high performance of a system simultaneously. General-purpose processors (GPPs) have more flexibility since they can execute various processes by changing software programs.

On the other hand, application specific integrated circuits (ASICs) have the advantage in performance due to their hardware implementation of applications. Reconfigurable devices exists between GPPs and ASICs. They attempt to realize the performance of ASICs using their hardware functions by mapping algorithms of target applications and, at the same time, make the flexibility of GPPs possible by reconfiguring their logic functions. There are, however, various problems about reconfigurable devices. Their problems include a killer application, configuration time, a tradeoff of their granularity and its associated issues, and so on.

A killer application is a major issue of researches on reconfigurable computing. There is no inevitability of using reconfigurable devices for a certain application since they lie between GPPs and ASICs and, therefore, there are many other suitable solutions for the application [2]. Although some reconfigurable devices show their performance in some application fields such as image processing and communications, their application fields are still limited. Therefore, it is necessary to seek the potentiality of reconfigurable computing systems.

Management of hardware resources is also one of the issues of reconfigurable computing systems in order to hide reconfiguration time of their components. In particular, static reconfigurable devices such as FPGAs require considerable time and reconfiguration during operation is not realistic. Therefore, as the solution for FPGA's slow reconfiguration time, partially reconfigurable FPGAs or dynamically reconfigurable devices have been proposed. However, if the size of target applications exceeds that of their configuration memory, they are required to replace their configuration data. In such a case, proper techniques to manage hardware resources and control configuration data are necessary to make more effective use of such systems.

Granularity is a significant issue posed to reconfigurable computing systems. Fine-grained architectures have the high flexibility since their function units are composed of small portion logics such as look-up tables. However, their area efficiency is low since interconnection wiring dominates their die area. On the other hand, coarse-grained architectures have advantages in performance since their basic unit comprises of arithmetical logic units. However, if the data width of basic units is different from that of applications' demands, their utilization efficiency becomes low. A reconfigurable 1-bit processor array [33] has been proposed as one of the answers to the granularity issue. However, it has not had its own design environment yet. Therefore, a mapping method is required to make clear the effectiveness of its approach.

In this dissertation, an utilization methodology of a reconfigurable computing system and two computer aided design algorithms for new appearing architectures whose function unit employs the reconfigurability are provided. First, reconfigurable computing systems are introduced in the following section. Then, an utilization scheme is presented for a reconfigurable platform using Event-Oriented Computing (EOC) as one of the possible solutions for application problems of reconfigurable hardware. Next, a new approach of online task placement for a partially reconfigurable FPGA is proposed for more utilization of hardware resources. A placement and routing algorithm for a reconfigurable 1-bit processor array is suggested. In the algorithm, the granularity issue is tackled and, therefore, its effectiveness should be cleared with mapping tools.

In Chapter 2, the possibility of utilizing a reconfigurable computing system is proposed. The performance improvement resulting from parallelism of applications is often spoiled due to a bottleneck of data communication between a core processor and reconfigurable logics. Therefore, the following conditions are indispensable for effective utilization of a reconfigurable hybrid system. First, applications need to be computation-intensive and highly parallel to extract their parallelism by reconfigurable logics. Second, the amount of data communication between a core processor and reconfigurable logics is small since it can become the bottleneck of the system. Event-Oriented Computing is a good candidate that meets these conditions due to its characteristics. By using EOC, a method for exploiting such a hybrid system is proposed to harness the reconfigurable logics effectively by avoiding the bottleneck. The system architecture is also presented and an evaluation of it is reported through an experiment. In the experiment, performance improvement results from the fact that the amount of data communication is small.

In Chapter 3, an efficient and effective algorithm is presented for online task placement with I/O communications in partially reconfigurable FPGAs. In a partially reconfigurable FPGA of the future, arbitrary portions of its logic resources and interconnection networks will be reconfigured without affecting the other parts. Multiple tasks will be mapped and executed concurrently in such an FPGA. Efficient execution of the tasks using the limited resources of the FPGA will necessitate effective resource management. A number of online FPGA placement methods have recently been proposed for such an FPGA. However, they cannot handle I/O communications of the tasks. Taking such I/O communications into consideration, a new approach to online FPGA placement is introduced. An algorithm is presented for placing each arriving task in an empty area so as to complete all the tasks efficiently. Two fitting strategies are developed to effectively handle I/O communications of the tasks. The experimental results show that properly weighted combinations of these and two other previously proposed strategies enable this algorithm to run very fast and make an effective placement of the tasks. In fact, the result shows that the overhead associated with the use of this algorithm is negligible as compared to the total execution time of the tasks.

In Chapter 4, a placement and routing algorithm for a reconfigurable 1-bit processor array is focused. A reconfigurable 1-bit processor array (1-bit RPA) has been proposed as an outcome of researches on reconfigurable devices. Its architecture is mainly composed of processor elements (PEs) that have bit-serial data paths. The interconnection networks among PEs are determined according to certain parameters, and its structure makes flexible mapping of applications possible. However, due to its unique wiring structure, a dedicated method is required to place and route target applications for the architecture. An efficient and effective placement and routing algorithm for a 1-bit RPA is presented. Preliminary experimental results using the algorithm are promising.

In Chapter 5, this dissertation is concluded and future work is described.

1.2. Reconfigurable Computing Systems

In this section, general ideas and some taxonomies of reconfigurable computing systems are introduced. According to granularity, there are two types of architectures: fine-grained and coarse-grained architecture. In addition, according to the configuration time, these architectures are further classified: static and dynamic.

Here, typical architectures in each category are explained. An FPGA is one of

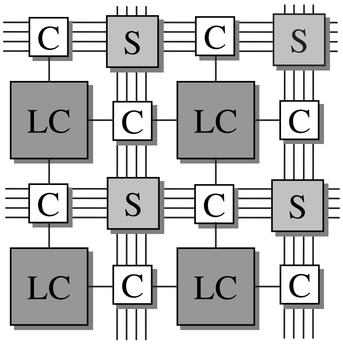


Figure 1.1. A block diagram of an FPGA

the typical reconfigurable devices that employ static fine-grained structure [6]. In Figure 1.1 shows a block diagram of the FPGA. In the figure, LC represents Logic Cluster (LC) that has multiple basic logic elements (BLEs). BLEs are mainly composed of look-up tables and registers that realize any logical functions. Switch blocks (SB) provide controls of intersection of buses and are shown as S. C is a connection block (CB) to connect LC and buses. Various functions are realized by controlling connections of SBs and CBs.

A design flow of FPGAs is also mentioned. FPGA's design flow is mainly composed of register transfer level (RTL) design, logic design, and placement and routing [41]. Logic design and placement and routing steps are automatically performed using a dedicated design tool that FPGA vendors provide such as ALTERA Quartus II [1] and Xilinx ISE foundation [49]. The design tool of FPGAs is well developed and many effective optimization schemes are proposed [5]. This promotes the wide use of FPGAs in the market [2].

Dynamically reconfigurable processor (DRP) [30] is the reconfigurable hardware that has a coarse-grained data path and supports dynamic reconfiguration during operation. Figure 1.2 shows a block diagram of a DRP and this basic structure is called tile. DRP is composed of DRP's tiles that are arranged in the form of array. The main components of DRP's tile are processor elements, state transition controller (STC), and memory array. PEs are basic arithmetic units whose bit width is 8 bits. STC is a simple sequencer to control the configuration data of a DRP. A PE has instruction memory for various operations. STC sends

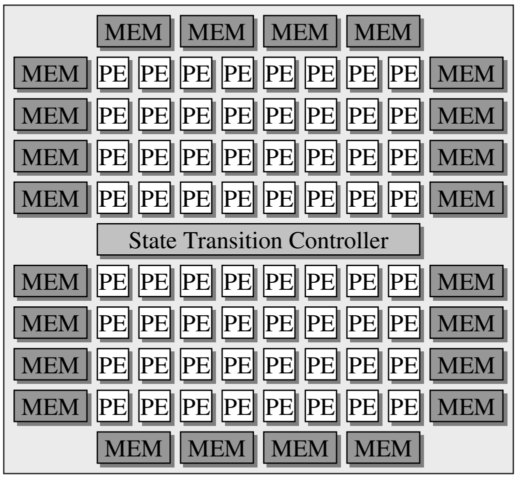

MEM MEM MEM MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
State Transition Controller
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM PE PE PE PE PE PE PE PE MEM
MEM MEM MEM MEM

Figure 1.2. A block diagram of a dynamically reconfigurable processor

instruction pointer to each PE and DRP can change its configuration dynamically by changing this instruction pointer.

Integrated design environment for a DRP has been provided [41]. Behavior design is described using C language and, if required, RTL design is done. The C description is converted to RTL design that is composed of finite state machine for STC and data path circuit for PE arrays. Then, placement and routing is performed and the compiler outputs object code for a DRP.

Chapter 2

The Possibility of Utilizing a Reconfigurable Computing System

2.1. Introduction

During the past decade, reconfigurable computing has been researched extensively as a new method to improve processor performance [11], [25]. In particular, reconfigurable hardware coexisting with a core processor is considered a good candidate for speeding up processor performance.

Such a hybrid system demonstrates its effectiveness in multimedia applications, data encryption, signal processing, communication fields and so on. The main idea behind improving the performance of such a solution is that reconfigurable logics process compute-intensive tasks of a target application instead of a general-purpose processor. Among proposed systems, dynamic reconfigurable processors, which can reconfigure their functions during operations, have received considerable attention.

One of the critical issues in a hybrid system composed of a core processor and reconfigurable logics is the communication between them. In many cases, the communication becomes the bottleneck of the system. Therefore, to utilize a reconfigurable computing system effectively, we need to avoid this communication bottleneck problem.

In this chapter, the exploitation method of such a hybrid system is worked on to harness the reconfigurable logics effectively. Effective utilization of reconfigurable logics is considered to need the following attributes. First, applications for reconfigurable computing are computation-intensive and highly parallel since this parallelism is exploited by reconfigurable logics. Second, data communication between a core processor and reconfigurable logics is relatively low, since in many cases it becomes the bottleneck of the system.

These attributes are conditions of tasks that should be assigned to reconfigurable logics. Through an actual application, a significant performance improvement will be shown. It results from the fact that the amount of data communication is small. Specifically, a performance improvement using Event-Oriented Computing that meets the conditions described above is demonstrated through the experiment. In the experiment, that small amounts of data communication lead to direct performance improvement is also shown in parallel with conducting an experiment on an unlikely situation in which the quantity of data communication is very large.

Among many applications, EOC is a good candidate that has the attributes required for the reconfigurable hybrid system. The processing of EOC includes many simple conditional evaluations that are frequently invoked and processed in parallel, and this parallelism can be extracted with relative ease. If the conditions become true, the corresponding actions are invoked in the process of EOC. However, cases where the conditions become true are quite infrequent due to the attributes of EOC. As a result, the number of processes invoked is small. It is therefore possible to reduce the necessary data communications between a core processor and reconfigurable logics; the extracted parallelism of EOC is not spoiled by the system's bottleneck.

A suitable architecture is also proposed to utilize a reconfigurable hybrid system in which EOC is leveraged to efficiently avoid communication bottleneck problems. In the processing of EOC, since the conditions vary according to the application implemented, effective use of dynamic reconfigurable systems is required for EOC to be successfully implemented. In fact, the experimental results reveal the potential of the method in comparison to software implementation.

The remainder of this chapter is organized as follows. Related work is covered in Section 2.2. Section 2.3 contains an explanation of the application model for utilization of a reconfigurable computing system. In Section 2.4, the system architecture is proposed, and then in Section 2.5, the system is evaluated. In Section 2.6, the conclusions and future work are described.

2.2. Related Work

In this section, work related to reconfigurable computing is discussed, focusing in particular on models of a core processor coexisting with reconfigurable logics, as with the model. In the field of reconfigurable computing [7], [10], [23], [26], [36], [42], many reconfigurable-hardware-based architectures and their applications have been proposed.

PRISC [37], [38] and Chimaera [21], [50] have a similar structure: each of them integrates a small number of reconfigurable units into its host processor's data path. In particular, Chimaera has a reconfigurable array consisting of 32 rows, and each row has the same number of logic cells as its host processor's bit width. By using nine shadow registers that are partial copies of its host processor's registers, it reduces the amount of data transfer between the reconfigurable array and its host processors, as the model does. However, since it improves the processor performance by the collapsing of instructions, it does not achieve a significant performance improvement.

OneChip [47], Garp [8], [22], REMARK [27], [28] and MorphoSys [39] also consist of a core processor and reconfigurable logics, though integration of both components is less tightly coupled. These reconfigurable architectures harness their reconfigurable resources to map long-running nested loops present in target applications. Therefore, these systems show good performance improvement with some applications. Although these proposed architectures have the capability to speed up processor performance in processing stream data like the encoding of motion pictures or communications processing, the bottleneck of the data communication between a core processor and reconfigurable logics still remains a key issue. Therefore, applications that need frequent communication between a core processor and reconfigurable logics are not suited for this purpose, since integration of a core processor and reconfigurable logics of those systems is weak.

On the other hand, in the architecture and application model, the unnecessary part of data communication is filtered out while a core processor and reconfigurable logics communicate with each other at every step. The system is suitable for applications that have the attributes of EOC. As mentioned earlier, extracting their parallelism is relatively easy, and the necessary amount of data communication between a core processor and reconfigurable logics is small due to application's characteristics. As a result, a reconfigurable computing system can be taken advantage of without spoiling its performance gain.

2.3. Possible Application for a Reconfigurable Computing System

As already stated, the following conditions as indispensable for possible application areas of the reconfigurable hybrid system.

1. Compute-intensive and highly parallel
2. Data communication quantity is small

EOC meets these conditions since its processing includes parallel conditional evaluations and only a small amount of data communications between a core processor and reconfigurable logics is necessary.

Figure 2.1 shows a block diagram of the proposed system architecture. In the figure, data communications between a core processor and the other components can become the bottleneck of the system. The amount of data communication between them is relatively small due to the attributes of EOC, making EOC a promising model for applications that effectively utilize a reconfigurable computing system. In the following sections, EOC is described in detail and some suitable applications are also mentioned.

2.3.1 Event-Oriented Computing as an Application Model

Here, EOC and its advantage are described. EOC is ruled by the ECA (Event Condition Action) model [40], [45], which is a sequence of processes derived from

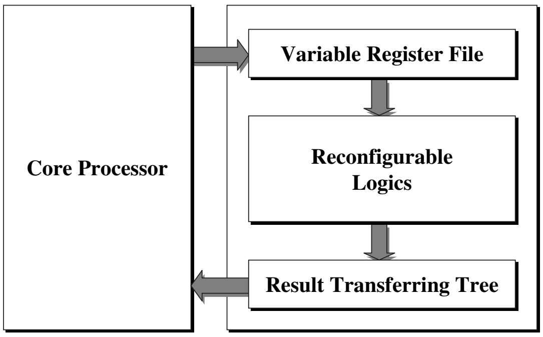


Figure 2.1. A block diagram of proposed architecture

the emergence of events (*Event*), the acceptance conditions (*Condition*), and the associated actions (*Action*).

The *Event* is a set of triggering occurrences that cause new events, the *Condition* is a set of expressions that have to be checked or evaluated when the event occurs, and the *Action* is a set of operations that are executed if the condition becomes true and, as a result, updates the *Event*.

The features of EOC are as follows.

- Flexibility
  - We can add new components to the flow with relative ease since application flow is controlled not in a sequential fashion but in an event-triggered fashion.
- Simplicity
  - Tasks are more understandable since each event and condition dominates target tasks.
- Robustness
  - In EOC, since strict ECA rules govern all of the components in application programs, application programs adopting this method become robust as indicated by the new programming paradigm called *Active Software* [24], [44].

The process of EOC requires parallel and frequently invoked evaluations of the conditions, and the ECA model governs this process. As a whole process, when events occur, the corresponding conditions are evaluated. If the conditions become true, the corresponding actions are executed. To process this ECA model of EOC by utilizing a core processor and reconfigurable logics, the following variables are associated with the system.

- Associated Variables N_i
- Triggering Events E_i
- Conditional Expressions C_i
- Updating Actions A_i

Each event features associated variables N_i. Triggering events E_i represent the occurrence of events, and Conditional expressions C_i represent conditions to be evaluated when events occur. Updating actions A_i denote functions to be activated when corresponding conditions become true. The basic flow of EOS is described as follows.

1. If associated variables N_i receive a change in value, the system treats them as the occurrence of events E_i.
2. Associated conditional expressions C_i are evaluated based on N_i.
3. If the conditional expressions C_i become true, the system processes the corresponding actions A_i that may update the associated variables N_i.
4. Associated variables N_i that are updated lead to triggering events E_i of the next step.

The process of EOC consists of these iteration steps, and each of the above steps is mapped to the system. Generally, conditional expressions C_i are relatively simple, and their number is large; furthermore, many of them do not become true. In such a case, as shown later, the reconfigurable logics of the system can be leveraged without any spoilage of performance gain due to the communication bottleneck. Although there is a case where many of the conditional expressions C_i become true, such cases are not dealt with here. To illustrate the model in more depth, an actual example is given in the evaluation in Section 2.5.

2.3.2 Applicable Applications of Event-Oriented Computing

In this section, suitable applications are outlined for the system dealing with EOC.

Active Software, originally proposed by R. Laddaga [24], is one of the applications that has the property of EOC. Watanabe et al. pushed forward the research on Active Software and proposed some practical methods and execution models in their paper [44]. According to their methodology, Active Software consists of *Active Functions*, and each Active Function has an activation condition that is dominated by the ECA model. The execution of Active Software is suited to the system because there are frequent evaluations of conditions occurring during its execution process.

Event detection is another potential application for the system. Here, event detection means detecting changes in an environment or exceptions from normal situations. An example is postural control in the field of robot engineering. When a robot walks, forces such as gravity or its own inertial forces act on it. In addition, the robot also receives ground reactions as counteractions from the land surface.

Most commonly, event detection calculates ideal gait patterns with computers while considering these forces, thus following these patterns. However, the ground surface may not always be smooth; there may be irregularities such as pebbles. Therefore, postural control systems require devices such as an angle meter or an articular angle controller. In such a system, the architecture can be applied to detect events by use of reconfigurable logics.

Computerized simulations such as physical or biological simulations are other potential applications for the system. Generally, simulations that involve changes in the environment include many evaluations of conditions. The system is capable of running such types of processing efficiently. In fact, in the following section the effectiveness of the model is demonstrated through simulations of *Artificial Life*.

2.4. Proposed Architecture Model

In this section, the system architecture is explained in detail.

2.4.1 Overview of Proposed System

A block diagram of the proposed system is shown in Figure 2.1. The system consists of four modules: a Core Processor (CP), a Variable Register File (VRF), a Reconfigurable Logics (RLs), and a Result-Transferring Tree (RTT). The CP is a central processing element that mainly performs application programs. The VRF is a module that stores variables associated with events. RLs are reconfigurable hardware parts. This module evaluates conditional expressions. RTT is a component that consists of tree-based buffer registers, transferring the results of evaluations from the RLs to the CP. In the following subsections, each module is described in detail.

2.4.2 Evaluation of Conditional Expressions

In this section, how to process Event-Oriented programs on the architecture is presented.

As mentioned in Section 2.1, in EOC, parallel and frequent evaluations of conditional expressions are inevitable.

Conditional expressions are evaluated in parallel based on associated variables. Each register of the VRF corresponds to the associated variable, and the CP transfers updated associated variables to the VRF. In general, reconfigurable hardware coexisting with a core processor has the crucial problem of communication between the core processor and the reconfigurable component. Therefore, in the system, the VRF runs independently of the RLs. That is, since the VRF **only** fetches associated variables from the CP and feeds the data to the RLs, it operates with a high-speed clock that is different from that of other modules. This permits fast data transfer between the CP and VRF. Figure 2.2 illustrates the communication between the CP and VRF.

RLs are processing elements that consist of reconfigurable hardware. The structure of the RLs is not critical here, and the optimal structure should be

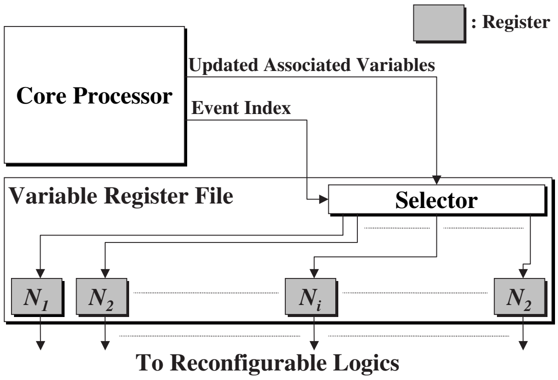


Figure 2.2. Variable register file

chosen based on the requirements of each application. In the example, a field programmable gate array is used for evaluation because of its flexibility. The conditional expressions are mapped to this module. If not all of the conditional expressions fit into resources, the RLs implement all of the conditional expressions by using context switches of virtual hardware. This module evaluates the conditional expressions in parallel with the associated variables from VRF. If the conditional expressions become true, it sets the corresponding flags to 1 and sends additional data to RTT. Here, a flag denotes the occurrence of updating action A_i. Namely, "$the\ conditional\ expression\ =\ true''$ means "$the\ corresponding\ flag\ bit\ =\ 1''$. Also, the additional data are the information that the CP uses to execute updating actions A_i.

2.4.3 Result Transferring Tree

Here, the structure of the RTT is explained. The RTT is used to achieve fast data transfer. In the EOC model, the events that become "$flag\ =\ 1''$ are relatively few as compared to all of the events to be evaluated. Therefore, in terms of parallelism, it is more efficient for the buffer registers of the RTT to receive all the flags at once than to receive the flags that are 1 sequentially after the processor checks all of them. In terms of data communication, it is efficient for the specific component to put the flags through a sieve and transfer only the flags of 1 to the CP.

The RTT is the component that realizes efficient data transfer between the CP and RLs. This component fetches issued flags and additional data to a queue, and then transfers them to the CP one by one. To fetch issued flags and data, the RTT propagates them on a tree-formed data path, reducing the number of steps needed to fetch flags and data to log(#Events) rather than #Events, where #Events is the number of events.

Figure 2.3 shows the architecture of the RTT, which consists of buffer registers and the control logic block. Buffer registers comprise the registers that store flags (flag registers) and the registers that store additional data (data registers). Flag registers store the flag bit to which the RLs output, while data registers store the additional data that the CP needs.

The control logic block consists of queue registers and a simple control circuit. The queue registers are also buffers for a flag bit and additional data, and the control circuit controls the transfer of them from the queue registers to the CP.

The structure of the RTT has the form of a binary tree, where data are propagated from leaves to the root. The RTT transfers flags and data to the CP according to the following steps.

1. The RLs set the results to the corresponding flag registers in parallel.

2. Buffer registers at the bottom level in the binary tree fetch the values of the flag register and the data register if the value of the corresponding flag is 1. A pair comprising a flag and the corresponding additional data are regarded as a data unit.

   A buffer register fetches such a pair from one of the two leaves. If both of the flag values are 1, the pair that has the smaller index is given priority.

3. The pairs in the buffer registers of each level are transferred to the next level's buffer registers if they are not occupied by other data. Each data set flows in the buffer registers from leaves to the top in such a way.

4. Finally, the values go into the queue registers in the control logic block.

5. The CP receives the data from the queue registers and executes the corresponding update action.

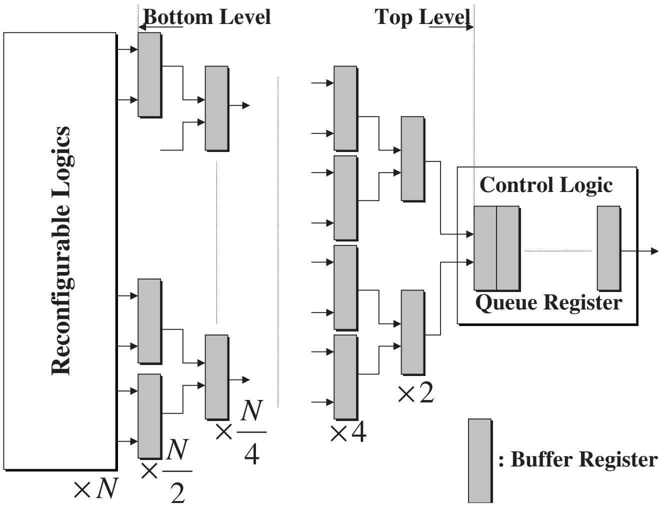
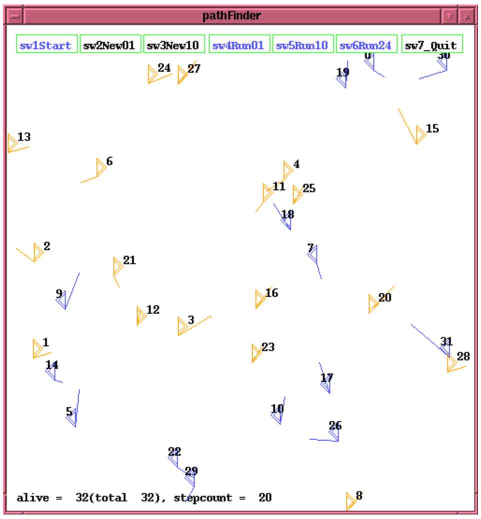


Figure 2.3. Result transferring tree

For example, let us consider the case where there are 32 conditional expressions. When the system sends the results of the 32 conditional expressions to the CP, it takes 32 clock cycles if the system sequentially checks which conditional expressions are true, and then sends the flags that are true. However, as repeatedly mentioned, the case where only a small number of conditional expressions become true is considered. If there are only four conditional expressions that become true, this can be handled more efficiently by using the RTT's tree-based architecture. After the latency of $log32 = 5$ and a few clocks, the necessary data can be sent to the CP.

2.5. Evaluation of the Proposed System

In this section, the architecture is evaluated, applying Artificial Life to the architecture as a typical example of EOC.



```
alive =  32(total  32), stepcount =  20
```

Figure 2.4. An image of artificial life simulation

2.5.1 Application of Artificial Life to the System

Artificial Life is an approach that simulates the world of life by using a computer to observe the behavior of living matter or its evolution and inheritance mechanism. In artificial life, each individual organism acts under various action rules and organisms interact with each other. The world of artificial life begins to change in mass as the result of this interaction. Since by following the process we can obtain a close-up view of the phenomenon of life, this approach has recently received attention as a new research method. In Figure 2.4, an image of an artificial life simulation is shown. In the figure, each triangular object is a unit of artificial life that interacts with the others. In this instance, the number of individual organisms is 32, half of which them are male and the others, female.

As noted earlier, in artificial life, each individual organism acts on its own accord, thus influencing the others. Accordingly, artificial life fits well with the ECA models of EOC. Simple artificial life is implemented in the architecture for evaluation. The details of artificial life are as follows.

- **Individual Organisms**
  - Each individual organism has only its location and the attribute of

  "male" or "female."
  - The number of total individual organisms is 32.

- **Interaction**
  - Basically, each individual organism moves around at random.
  - If there is another individual organism within a certain distance, the two individual organisms interact with each other.
  - If they have opposite sexuality, the male individual chases the female one. In this regard, however, if they get too close to each other, they move in the opposite direction.
  - If each of them has the same sexuality, they move in the opposite direction.

The artificial life described above corresponds to the architecture model as follows. Here, i is each individual's index: (0 <= i <= 31).

- Associated Variables N_i
  - Associated variables N_i store each individual organism's index and its location.
- Triggering Events E_i
  - Triggering events E_i correspond to the updating of the individual organism's position.
- Conditional Expressions C_i
  - Conditional expressions C_i are the conditions that represent whether other individual organisms exist within a certain distance.
- Updating Actions A_i
  - If other individual organisms exist within a certain distance, conditional expressions C_i become true. Then, the flag bit corresponding to the C_i is set to 1, which will invoke A_i. A_i updates N_i, i.e., the location

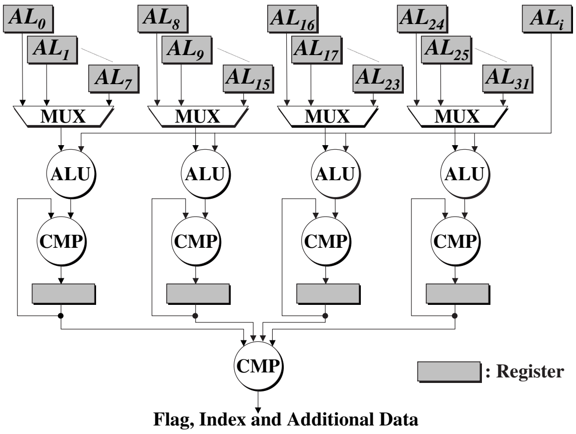


Figure 2.5. Comparator module of artificial life

of the corresponding organism. To update the location of each individual, the information on the closest organism is needed. Therefore, the index of the closest organism, which is called "additional data" in the model, is transferred to the CP with the flag bit. The corresponding updating action A_i is then processed in the CP according to distance and sexuality.

2.5.2 Implementation of Artificial Life

In this section, how artificial life is implemented in the architecture is explained. In the implementation, the CP and the other components are connected with a general communication protocol, say, PCI [35]. For the evaluation, an FPGA, ALTERA Stratix EP1S80F1508C6, is used as the RLs' structure. However, since the CP and the communication bus between the processor and the other components are not implemented, the communication latency is estimated from the specifications of the PCI, and use this latency for the evaluation.

At the beginning, the CP gives start positions to each unit of artificial life at random. Next, it sequentially transfers the data for the artificial life to the VRF, which feeds these data to the RLs. The RLs has 32 copies of a comparator module to manipulate these data in parallel. Each comparator module consists of some arithmetic logical units as shown in Figure 2.5.

In Figure 2.5, AL represents the artificial life data in the register of the VRF, where ALU represents the calculation of the Manhattan distance of each artificial life unit, and CMP represents the operation that compares the Manhattan distances and takes the lower one. Each comparator module executes the calculation of the Manhattan distances between one and the other 31 artificial lives and detects the closest artificial life unit.

Because the comparator module has four calculation blocks, it can calculate the Manhattan distance of up to four other artificial lives in parallel. For example, the first step is to compare AL_i with AL_0, AL_8, AL_{16} and AL_{24}. The second step is to compare AL_i with AL_1, AL_9, AL_{17} and AL_{25}. The module iterates operations like this up to eight times. After an iteration, it narrows the results down four and selects the lowest one. As a result, the positional relationships of all of the artificial life units can be obtained.

After that, the RLs check whether the Manhattan distance is within a certain threshold by using this positional relationship and, if so, issues a flag and outputs the corresponding additional data. These data are transferred through the RTT to the CP. After the processor receives the data, it updates the corresponding artificial life unit's position and transfers the updated data to the VRF. By iterating this process, the appearance of Artificial Life can be observed.

2.5.3 Experimental Results

In this section the experimental results obtained with the architecture are shown. Table 2.1 shows the number of logic cells and the clock frequency of each module. However, it should be noted that the values of clock frequency of the VRF and RTT are estimations. With continued advances in technology, these values are expected to be able to be implemented and, therefore, it can be considered that using these values is reasonable. The other values of the table are observed ones. The latency between CP and RTT depends on how many conditions become true. For reference, the cases where 8, 16, and 32 conditions become true are shown in Table 2.2. Table 2.3 shows the time needed to process the aforementioned operations.

For comparison, the same process is implemented using C language on the systems of an Intel Pentium III processor running at 1.4 GHz and an Intel Xeon processor running at 2.4 GHz. Their operating system is GNU Linux OS.

In the architecture, data flow is as follows.

- $CP \rightarrow VRF \rightarrow RLs \rightarrow RTT \rightarrow CP$

To estimate the total processing time on the architecture, it is necessary to know the communication time between the CP and VRF and between the RTT and CP. The execution time on the RLs and the RTT is also needed. Furthermore, the time used to update variables on the CP is needed and, next, this value is estimated.

In software implementation, the total processing time of an iteration is 31240 ns in the Pentium III system and 23766 ns in the Xeon system, respectively. In the Pentium III system, the time used to update variables in the iteration is about 965 ns. Here, that the CP's clock frequency is the same as the Pentium III system's is assumed. Therefore, the time used to update variables on the architecture is about 965 ns.

Next, the communication time between CP and RTT is estimated. That the VRF and RTT are connected to the CP by general communication protocols is assumed. One example of them is the peripheral component interconnect (PCI). According to the specifications of the PCI, its bus width is 32 $bits$ and its clock frequency is 33 MHz. Therefore, the data latency per data unit is about 30 ns ideally. However, considering the overhead of protocol communication, three patterns of the data latency are used. The first case is the ideal one: Its data latency is about 30 ns. The next case is five times slower than the ideal case. Its data latency is about 150 ns. The last case is the ten times slower than the ideal case. Its data latency is about 300 ns. These figures are summarized in Table 2.2.

Here, the communication time between the CP and VRF is estimated. The VRF receives the updated variables and there are 32 data that need to be transferred. From the above three patterns, the latency between the CP and VRF is 960 ns, 4800 ns, and 9600 ns in each case.

Next, the execution time on the RLs is shown. The VRF, RLs, and RTT are implemented in a single chip. The RLs run at 61.5 MHz and their clock cycle is

Module	Clock Frequency	Number of Logic Cells
VRF	200 MHz	902 $Cells$
RLs	61.5 MHz	24370 $Cells$
RTT	80 MHz	1104 $Cells$

Table 2.1. Evaluation of each module

16.26 ns. They needs one clock cycle for acknowledgment and eight clock cycles to calculate the Manhattan distance, plus one more clock cycle to narrow down the four results. Therefore, its processing time is about 162.6 ns.

Here, the execution time on the RTT is shown. The RTT runs at 80 MHz and its clock cycle is 12.5 ns. In this implementation, because the RTT has four queue registers, it needs one clock cycle to receive the acknowledgment from the RLs and eight clock cycles to transfer the first result from the top leaves to the end of the queue registers. Its processing time is about 112.5 ns.

Next, The total execution time on the architecture is presented. It is obtained as a summary of the above-mentioned processing time. The summary is shown in Table 2.3, but it should be noted that the latency of updating the queue registers of the RTT may overlap the CP's processing time.

As a result, in the eight-hit ideal case, the model achieves a processing time about 14.39 times faster than software implementation when using a Pentium III processor. In this implementation, the average hit rate is about 16.7 hits; therefore, in the 16-hit ×5 case, the model is about 3.70 times faster than a Pentium III system. In the worst case (32-hit ×10), the model's processing time is about 1.16 times faster than software implementation when using the Xeon processor. These results show that the communication time greatly affects the total processing time, and the method produces the intended effect.

2.6. Conclusions

In this chapter, a utilization method for a reconfigurable computing system has been shown by taking advantage of Event-Oriented Computing. A system archi-

Event Hits	×1	×5	×10
8 *hits*	240 *ns*	1200 *ns*	2400 *ns*
16 *hits*	480 *ns*	2400 *ns*	4800 *ns*
32 *hits*	960 *ns*	4800 *ns*	9600 *ns*

Table 2.2. Latency between the core processor and the result-transferring Tree

Model	×1	×5	×10
8 *hits*	2170 *ns*	7240 *ns*	13240 *ns*
16 *hits*	2410 *ns*	8440 *ns*	15640 *ns*
32 *hits*	2890 *ns*	10840 *ns*	20440 *ns*
Pentium III	31240 *ns*		
Xeon	23766 *ns*		

Table 2.3. Performance evaluation

tecture suited for the model has also been proposed.

As is often the case with reconfigurable logics that are combined with a core processor, the communication between them can become the bottleneck of the system. As a result, the parallelism extracted to the reconfigurable logics is not fully exploited.

On the other hand, EOC is an application model that is not affected by this problem due to its specific characteristics. First, in the EOC processing, there are many simple conditional evaluations and they are parallel and frequently invoked. As a result, the performance gain can be derived from the parallel processing of the conditional expressions by using the reconfigurable logics. Second, this performance gain would not be spoiled by the communication bottleneck problem when the number of conditional expressions that become true is small.

In the architecture for dealing with EOC, the processing and the data communication are overlapped to reduce the communication bottleneck. In addition, tree-based transferring buffers of the architecture reduces the latency to transfer the results of conditional evaluations. Also, through the experiment using

Artificial Life, the conditions have been revealed for utilizing a reconfigurable computing system effectively and demonstrated the potential capability of the system.

As future work, the architecture need to be implemented at greater depth and investigate other applications for the system.

Chapter 3

An Efficient and Effective Algorithm for Online Task Placement with I/O Communications in Partially Reconfigurable FPGAs

3.1. Introduction

In the late 1990's, research efforts exploded in the area of dynamically reconfigurable computing systems. Notable among them were the projects centered around Dynamically Programmable Gate Arrays (DPGAs) [12] of MIT and Plastic Cell Architecture (PCA) [31], [32] of NTT. Influenced by the results in these and other research projects, the technology of Field Programmable Gate Arrays has rapidly advanced and some type of partially reconfigurable devices such as Xilinx Virtex II Pro [48] are now available on the market.

With continued advances in technology, partially reconfigurable FPGAs of the future are expected to be able to dynamically reconfigure arbitrary portions of their logic resources and interconnection networks 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. 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 freed, 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 [17], 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 a lot of empty areas exist. This necessitates effective FPGA resource management. Such placement management is provided by a so-called *reconfigurable operating system* (ROS) [17], [29], [46]. 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 in 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 [13], [15].

On the other hand, in the online placement, the sequence of the task flow is known only at run time. Thus, the system needs to quickly find an empty area for each arriving task to reduce the overhead time. Consequently, the balance between time and quality becomes more important. Several online task placement algorithms have recently been developed for a simple model of rectangular tasks [4], [19], [20], [43]. However, this task model ignored the impact of the use of communication channels between the tasks and the I/O elements located on the periphery of the FPGA.

In this chapter, a new model for online task placement in partially reconfigurable FPGAs is introduced. Although it is based on the rectangular task model, the model considers I/O communications of the tasks. An online task placement algorithm which uses a weighted combination of four fitting strategies presented. to find a most preferable empty area for each task. Two of the strategies are new and explicitly reflect on I/O communication information of each empty area. The experimental results show performance improvements over the conventional approach. In addition, the running time of this algorithm is shown to be extremely short.

In the next section, some of the earlier work on reconfigurable computing systems and their task placement methods are reviewed. The model for a reconfigurable computing system is described in Section 3.3. In Section 3.4, the two new fitting strategies and the online task placement algorithm are explained. Then, a set of evaluation criteria is presented and the simulation results are shown in Section 3.5. Section 3.6 summerizes this chapter and describes some future work.

3.2. Previous Work on Task Placement

To the best knowledge, Bazargan, et al. [4] 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. [43] 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 [18], [19], [20] 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) [4], which may overlap with each other [20].

All of the algorithms mentioned above are based on the rectangular task placement model [4]. However, this model ignored the impact of the use of communication channels between the tasks and the I/O elements located on the periphery of the FPGA. In the next section, a new model for online task placement on partially reconfigurable FPGAs is introduced. Although it is based on the rectangular task model, the model considers I/O communications of the tasks. It should be noted that although the PCA model mentioned earlier, used a different architecture than ours, communications between the tasks and the I/O elements as well as among the tasks were important aspects of their reconfigurable systems

[31], [32].

Before proceeding to the presentation of the model, some other task placement work based on models other than the rectangular task model is described. Gericota, et al. [16] proposed a configurable logic block (CLB) based management of logic resources. Their method rearranges CLBs on an FPGA surface at run time so as to make an empty area sufficiently large to accommodate a task. Compton, et al. [9] proposed a different approach in which task relocations and transformations are applied to overcome the fragmentation problem.

3.3. Model of a Reconfigurable Computing System

The model for a reconfigurable computing system is described in detail. Each aspect of the model is presented in a separate subsection.

3.3.1 System Model

Figure 3.1 depicts the system model consisting mainly of a host CPU, a shared memory, a ROS, and a partially reconfigurable FPGA. The basic units of the FPGA resources are CLBs arranged in the form of a two dimensional array. 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.

The flow of processing of the online FPGA placement is shown in Figure 3.2. 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 the 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. For example, a smaller task may be scheduled for execution before the other larger tasks when there are no empty areas of sufficient sizes. After that, the placer searches the list of empty rectangular areas

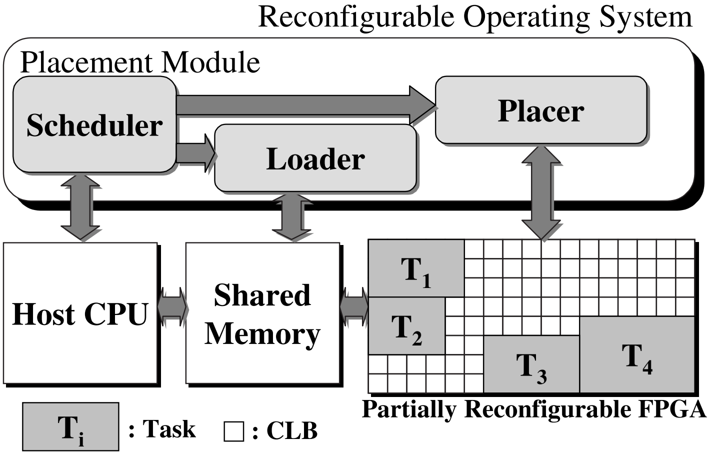


Figure 3.1. System model

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.

3.3.2 FPGA Model

As mentioned earlier, the FPGA consists of CLBs uniformly arranged in the form of a two dimensional array. They are used as basic logic elements in such FPGA products from Xilinx as its Vertex series [48]. By reconfiguring CLBs, digital circuits are implemented by an FPGA. In the FPGA model, that each CLB can be reconfigured independently is assumed. 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.*

Note that so-called "partially reconfigurable" devices currently available on the market do not yet have the capability of configuring any single CLB at anytime. With proper technological advancement, an enhancement of this capability in the future is expected. With this in mind, much work has already been done [4], [20], [43].

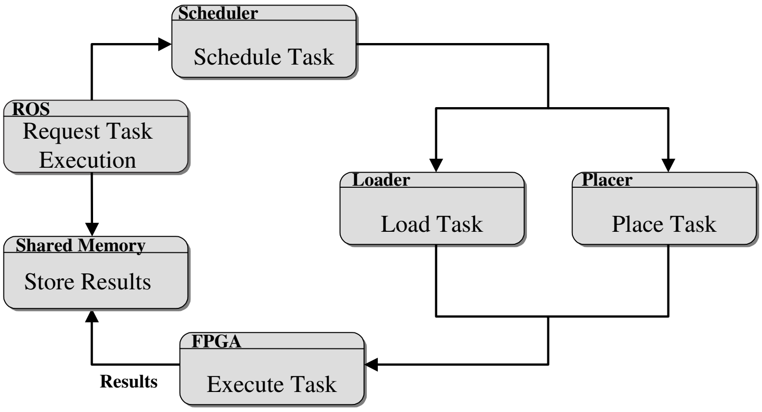


Figure 3.2. Flow of processing

As depicted in Figure 3.3, 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 the model, that communications take place between a task and an I/O element located on the periphery of the FPGA is assumed. The I/O elements then manage communications to external components such as the host CPU and the shared memory. In other words, no direct communication between a pair of tasks takes place. If a task communicates with another task frequently, they are merged to form a larger task. In the case of infrequent communications between tasks, communication messages are to be exchanged by way of the shared memory.

The 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. Therefore, all of the communications are done through communication channels and I/O elements. Note that communication resources occupied by a task are not used for I/O communications of the tasks since each task is a hardware macro. The usage of communication resources outside the tasks are expected to be much lower than that inside each task. Thus, even if unlimited numbers of communication channels are available, the amount of communication resources actually used is to remain within a limit of a practical

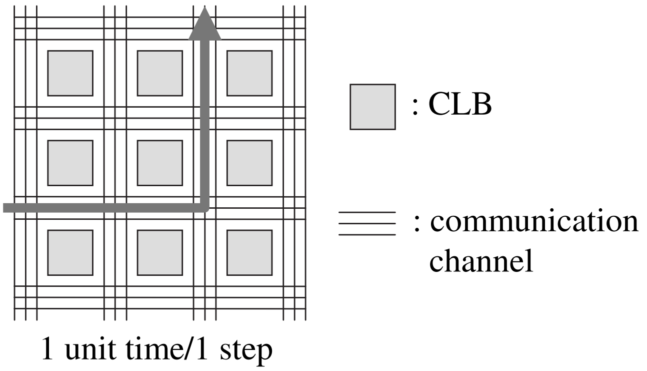


Figure 3.3. Communication delay

value.

In the model, communication costs are considered in the following manner. As shown in Figure 3.3 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. For example, if a communication path is represented by the gray arrow of Figure 3.3, it takes 4 units of time to propagate the signal.

3.3.3 Task Model

Now the assumptions on some features of a task in the model are described. The shape of a task is rectangular. An actual shape of logic circuits to implement each task may not always be rectangular. However, for ease of management, all tasks are regarded as being of a rectangular shape, granted that some areas might be wasted.

In the FPGA model, that different clocks may propagate to each CLB is assumed. 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) × (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

3.3.4 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. The placer takes some conditions into consideration when it searches for feasible areas for a task. The size of the task is such a main condition and, clearly, the placer cannot place the task in an area which is smaller in size than the task. It also looks for areas to avoid fragmentation of the FPGA resources as much as possible. Additionally, in the model, the placer handles I/O communications of the tasks, which may become a tradeoff to fragmentation of the FPGA surface. Thus, the objective of the placer is to place tasks online in the FPGA in such a way that the degree of fragmentation and the speed of I/O communications are balanced.

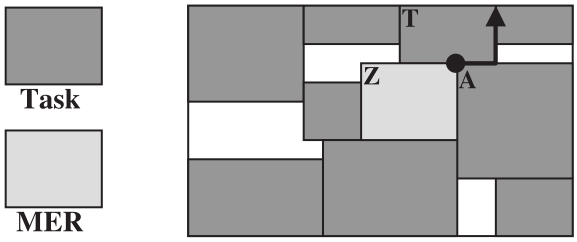


Figure 3.4. Task boundary = path

3.3.5 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, the boundaries of the tasks is used as much as possible. In Figure 3.4, 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 3.4 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 3.5 shows a case in which no task boundary is part of a communication path. The figure illustrates the task placement right after Task T of Figure 3.4 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 The empty area X has been lost due to the path of the empty area Z through the empty area X. Therefore, the model uses task boundaries for a communication path of each empty area as much as possible.

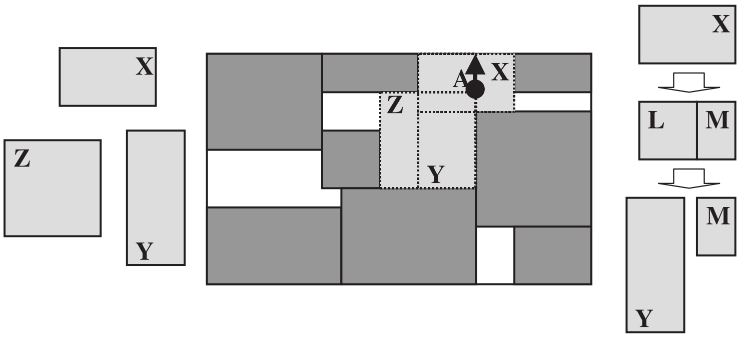


Figure 3.5. Task boundary $\neq$ path

3.3.6 Evaluation Methods

Two methods of evaluation of an FPGA task placement are presented, depending on whether the tasks are independent or dependent. An execution process of a task is depicted in Figure 3.6, 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 3.7 shows an example. The term *average waiting time* is

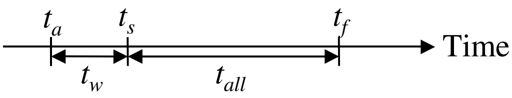


Figure 3.6. Execution process of a task

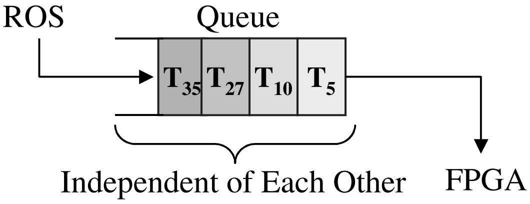


Figure 3.7. Independent tasks

defined for n independent tasks as

$$\overline{t_w} = \frac{1}{n}\sum_{i=1}^{n} t_{w_i} \tag{3.1}$$

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, The term *average overhead time* is defined as

$$\overline{t_{oh}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{t_{c_i} \ + \ t_{w_i}}{t_{e_i}}\right) \tag{3.2}$$

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

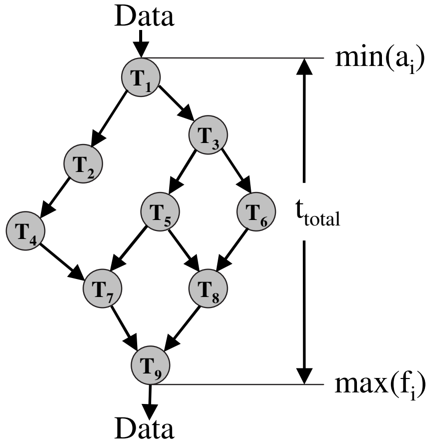


Figure 3.8. Dependent tasks

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

$$t_{total} = \underset{j}{max}(t_{f_j}) \; - \; \underset{i}{min}(t_{a_i}) \tag{3.3}$$

Figure 3.8 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 can execute an application.

3.4. Task Placement Algorithm

This section starts with an overview of the task placement method that handles I/O communications of the tasks. The main modules of the placement engine are a scheduler, a loader, and a placer. Among them, the placer plays a major role. Thus the four components of the placer: an FPGA surface manager, an I/O routing engine, an empty area manager, and a fitter are focused on. The surface manager provides a way to obtain empty areas on the FPGA surface and is described in Section 3.4.1. The routing engine provides an algorithm for creation of routing information for the empty areas obtained and is presented in Section 3.4.2. The empty area manager gives a method of management of the empty areas and is explained in Section 3.4.3. The fitter selects a suitable area from a group of empty areas by use of fitting strategies that are given in Section 3.4.4.

The relationship among these components is briefly described. 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.

3.4.1 Management of FPGA Surface

The first component of the task placer is presented. An efficient management of the FPGA surface is needed for full utilization of partially reconfigurable FPGAs. For this, a modified version of the method proposed by Handa, et al. [20] is used. A brief explanation of their method follows.

This part starts with the following definition [20].

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

MERs are used to manage the empty areas on an FPGA surface. Examples of MERs are depicted in Figure 3.9, where dark and light gray rectangles represent tasks and MERs, respectively, and some of the MERs overlap with each other.

Now how to find an MER on an FPGA surface is explained. A data structure, called *area matrix*, is used to store MERs. As shown in Figure 3.10, each cell in the area matrix corresponds to a CLB in the FPGA. Each cell stores a value, called the weight of the cell. The polarity of a weight indicates the state of the corresponding cell. A positive and a negative weight, respectively, corresponds to an empty and occupied cell. The value of a positive weight of a cell P gives the number of contiguous empty cells located at and above the cell P in the column

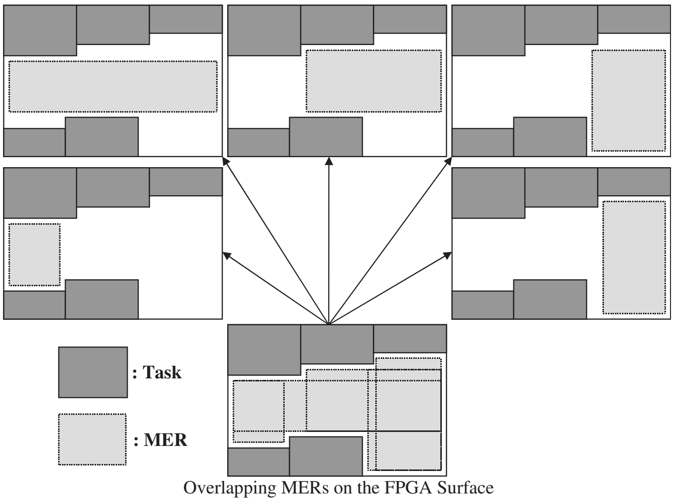


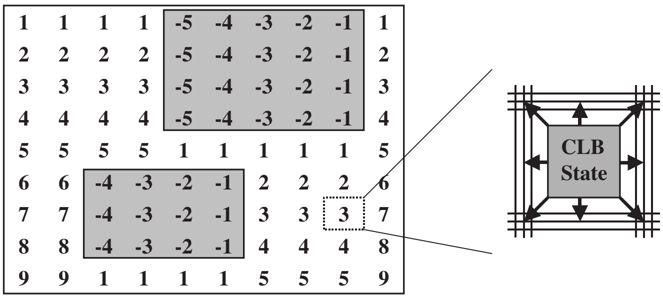
Figure 3.9. Overlapping maximal empty rectangles

in which P resides. The value of a negative weight in a cell N shows the number of contiguous cells such that (1) they are located to the right of the cell N in the row in which N resides and (2) they accommodate the corresponding part of the task that also occupies N. In the new model, the routing information from each task to an I/O element is handled. Hence the state of the communication channels around each CLB is added to its corresponding cell of the area matrix as depicted in Figure 3.10.

The following definitions are needed in order to describe a way to find an MER.

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.*

Rows of the area matrix are scanned and a staircase is made as shown in Figure 3.11. Since each cell P of the area matrix keeps the number of empty cells at and above P in its column, A staircase can be constructed by scanning rows. For example, in Figure 3.11, staircase *S* with the origin *O* can be obtained by scanning the row just above the top horizontal boundary of task *T*. In this way, MERs *OB*,



1	1	1	1	-5	-4	-3	-2	-1	1
2	2	2	2	-5	-4	-3	-2	-1	2
3	3	3	3	-5	-4	-3	-2	-1	3
4	4	4	4	-5	-4	-3	-2	-1	4
5	5	5	5	1	1	1	1	1	5
6	6	-4	-3	-2	-1	2	2	2	6
7	7	-4	-3	-2	-1	3	3	3	7
8	8	-4	-3	-2	-1	4	4	4	8
9	9	1	1	1	1	5	5	5	9

Figure 3.10. Area matrix

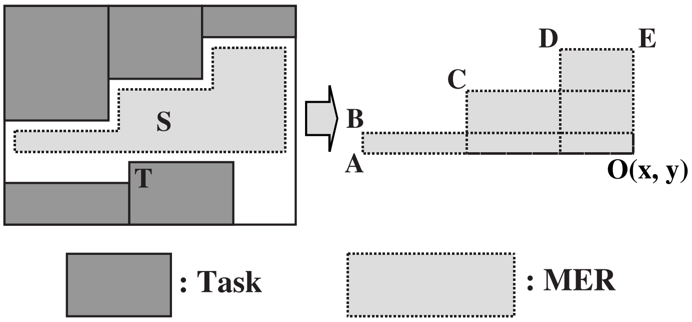


Figure 3.11. Staircase

OC,and *OD* are obtained. It has been proven that each staircase always rests on the top horizontal boundary of an already placed task [20]. 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 the conventional model, no I/O communication is considered and hence a staircase of the type shown in the left half of Figure 3.12 is obtained. In the model, however, the existence of such an I/O communication path for a task may necessitate the partitioning of a staircase into two as depicted in the right half of the figure. Note that after the partitioning, the left part of the original staircase will be expanded to be an MER in that region.

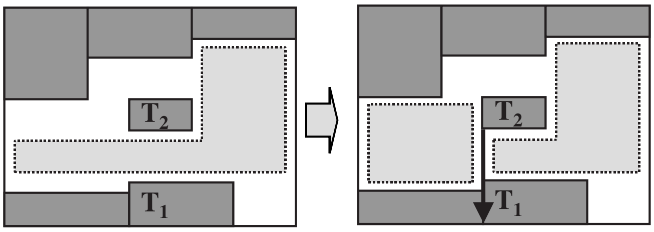


Figure 3.12. Partitioning of Staircases

As the example of this figure shows, the status of each communication channel need to be checked when staircases are created by scanning rows of the area matrix. If a vertical communication channel is occupied, the CLB just to the left of this channel becomes the origin of a staircase. The construction of a new staircase begins from the CLB just to the right of the occupied channel. Likewise horizontal communication paths affect the number of rows to be scanned. Rows just above each of such paths need to be scanned.

3.4.2 I/O Routing Algorithm

The second component of the task placer is an I/O routing engine. A greedy algorithm of construction of a communication path from a task to an I/O element located on the periphery of an FPGA is presented. It also determines the length of the path denoted by L. Vertex A is used to represent a base point of the path and letter E to denote one of the four boundary edges of an FPGA core that is closest from this vertex A. Initially, this base point A is selected as a vertex such that the sum of the shorter distances from it to a horizontal and a vertical edge, respectively, of the FPGA core.

1. Among the four vertices of an MER, find an initial base point A and an edge E as described above.

2. **if** A is already on E **then** output $L = 0$ and terminate.

3. **while** A is not on E
   begin perform the following operations:

```
4. case 1: ($A$ is border on a task in the direction to $E$ and $A$ is not any of the four corner vertices of the task.)
   if $E$ is a horizontal edge of the FPGA
   then begin
   if $A$ is closer to the right vertical edge of the FPGA
   then find the right vertex of the task and name it $a$.
   else find the left vertex of the task and name it $a$.
   Add to $L$ the distance from vertex $A$ to $a$ and move $A$ to $a$.
   end
   else perform similar operations for the case of a vertical edge $E$ of the FPGA.

5. case 2: ($A$ is border on a task in the direction to $E$ and $A$ is a vertex of the task.)
   Find the other end of the task in the direction to $E$ and name it $a$. Add to $L$ the distance from vertex $A$ to $a$ and move $A$ to $a$.

6. case 3: ($A$ is border on two tasks in the direction to $E$.)
   Determine the task out of two that has a shorter edge in the direction to $E$. Find the other end of the shorter edge and name it $a$. Add to $L$ the distance from vertex $A$ to $a$ and move $A$ to $a$.

7. case 4: ($A$ is not border on any task in the direction to $E$.)
   Find a point on the MER's edge or on the task's edge which is the closest to $E$ such that it is border on a task. Name it $a$. Add to $L$ the distance from vertex $A$ to $a$ and move $A$ to $a$.
   if there is no point that is border on any task in the direction to $E$.
   then make a path to $E$ until the point where there is a task. Name the point on the task's edge $a$. Add to $L$ the distance from vertex $A$ to $a$ and move $A$ to $a$.

8. end

9. Output $L$ and terminate.
```

Note that in the above method only four case statements are needed since A is not border on more than two tasks. It should also be noted that in its execution,

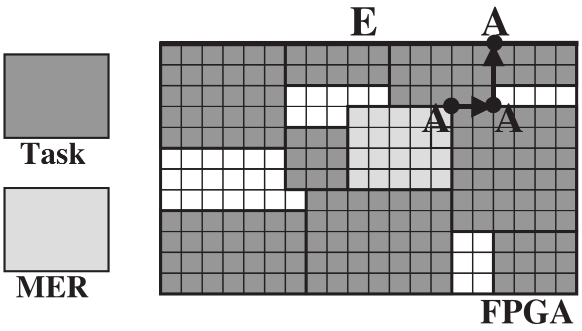


Figure 3.13. Search for an I/O communication path

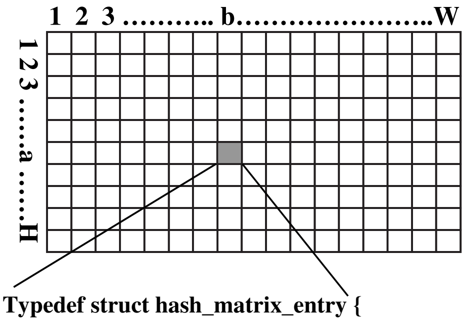
if a path reaches an edge of the FPGA, the algorithm terminates at that point.

Using an example from Figure 3.13, how the routing algorithm works is explained. At the beginning the algorithm selects as A the vertex at the upper-right corner of the MER and as E the upper horizontal edge of the FPGA. A is border on the task immediately above (see Case 1). Since A is closer to the right vertical edge of the FPGA, the algorithm finds as a the lower-right end of the task and adds to L the distance from A to a, which becomes a new vertex for A. This makes $L = 2$. The new A is closer to the upper edge of the FPGA, which is kept as E. Now A is border on the same task in the direction to E (see Case 2). The algorithm selects the other end of the vertical edge of the task as the new A and adds the distance from the old vertex for A to this end vertex to L. Thus, $L = 5$ is obtained. The algorithm will terminate with the outputs of the reverse L-shape path and its length of $L = 5$.

3.4.3 Management of MERs

This part turns the attention to the third component, an empty area manager, which also plays an important role in the delivery of good task placement. It starts with the following definition of the data structure introduced by Walder, et al. [43].

Definition 3 *A hash matrix is an array of size $h \times w$, where the parameters h and w denote the numbers of rows and columns, respectively, of the CLBs in an*



```
Typedef struct hash_matrix_entry {
  list of  Maximal_Empty_Rectangles (a×b);
  Maximal_Empty_Rectangle* free_mer_pointer;
} Hash_matrix_entry;

Hash_matrix_entry Hash_Matrix[Height][Width];
```

Figure 3.14. Hash matrix

FPGA.

Figure 3.14 shows an example of this data structure in detail. $Entry[a][b]$ of a hash matrix is associated with MERs of size $a \times b$. Namely, a pointer to a list of all such MERs is stored in the location of $array[a][b]$. As described below, an additional entry is needed in the same location. This entry points 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.

3.4.4 Fitting Strategies and Cost Functions

Finally the fourth component, the fitter, of the task placer is described. 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) Path 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. [17]. Strategies (2) *I/O Oriented* and (3) *Path Duration* are introduced to deal with I/O communications of tasks in the model. For each strategy its associated cost function is defined.

Strategy 1: *Best Fit*
Under this strategy, from the pool of MERs, a smallest MER that can accommodate the arrived task is selected. 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.

$$Cost_{BF} \;=\; \frac{S_{MER}}{S_{task}} \tag{3.4}$$

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

Strategy 2: *I/O Oriented*
Under this strategy, an MER that can provide faster communication of data for the arrived task is selected. Each task has a parameter b_{com} that indicates the amount of its communication bits. Each MER holds a parameter d_{path} which denotes the number of steps of its communication path to an I/O element. Let t_{unit} be the delay time for a task to route its data through a 1 step communication channel. Let w_{band} denote the number of communication channels available per task. The cost function for the I/O Oriented strategy is now defined as.

$$Cost_{IO} \;=\; d_{path} \;\times\; t_{unit}\frac{b_{com}}{w_{band}} \tag{3.5}$$

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, a new fitting strategy in the model is needed. The duration of path of an MER is defined as the average over the durations of all tasks that are border on the path of the MER. Under the new strategy, an MER whose duration of path is as close to the duration of the arrived task as possible is selected.

An I/O communication path of an MER passes along task boundaries. So a parameter t_{path} is defined 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. The cost function for Path Duration is defined as follows.

$$Cost_{PD} \;=\; \begin{cases} 0 & (t_f \;\leq\; t_{path}) \\ \frac{t_f \;-\; t_{current}}{t_{path} \;-\; t_{current}} & (t_f \;>\; t_{path}) \end{cases} \tag{3.6}$$

If $t_f \;\leq\; t_{path}$, 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 a selection of an MER from areas with more tasks allocated is tried so as to prevent other less crowded areas from being fragmented. Handa, et al. [17] introduced a parameter to quantify the fragmentation of an FPGA surface and provided detailed discussions on this strategy. For completeness, their definition of the cost function for Fragmentation Aware is simply given. 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.

$$TF \;=\; \frac{\sum_C TFCC}{S_{MER}} \;\times\; 100 \tag{3.7}$$

$$Cost_{FA} \;=\; \frac{1}{TF} \tag{3.8}$$

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

Now the total cost function is defined 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.

$$\begin{aligned} Cost_{ALL} &= \alpha \cdot Cost_{BF} + \beta \cdot Cost_{IO} \\ &\quad + \gamma \cdot Cost_{PD} + \delta \cdot Cost_{FA} \end{aligned} \tag{3.9}$$

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.

3.5. Evaluation of The Placement Engine

In order to evaluate the effectiveness of the four fitting strategies described above and their combinations, Simulations are conducted for the case of an FPGA with 96 × 64 CLBs. There is no intention to claim that future FPGAs with partial reconfigurability will be of this size. It is adopted simply because (1) one of Xilinx's current FPGAs, Virtex XCV-1000 is of this size, and (2) the previous studies most relevant to the work used FPGAs of the same size for their simulations [17], [18], [43]. 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. The uniform distribution in task size from a maximum to a minimum for each of the ten sets is observed.

By setting appropriate parameter values, three different task sets, called a small, medium, and large task set, are created. Note that the sizes of the tasks affect fragmentation of the FPGA surface. With these task sets of different sizes, the effect of each of the fitting strategies will be observed. 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.

Before proceeding to the simulation results, the reader is reminded again of the use of randomly generated data for evaluation of the algorithm. This has always been done in the past simply because it is impossible to generate real data for future technological advancement [4], [20], [43].

Most of the previously proposed algorithms in the conventional model mainly used Strategy 1, *Best Fit*. Thus, the case in which only Strategy 1 is used, namely, $\alpha = 1$ and $\beta = \gamma = \delta = 0$ corresponds to the conventional method is assumed.

This case is designated 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, the online placement algorithm is run with certain strategies emphasized over the others. The total execution time, average waiting time, and average overhead time are measured. In order to observe the performance of the new fitting strategies proposed for the model with I/O communications, the values obtained in these three measures are divided by their corresponding values for Case 1 (i.e., the conventional method). In each figure to follow, These fractional values for these four cases are shown. 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, The values for parameters α, β, γ, and δ are carefully selected. In particular, the simulation results indicated a good performance improvement by Strategy 2 of *I/O Oriented* for each of the task set categories. The case of these new parameter values for the total cost function is designated as Case 5.

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

In addition, a default case in which the parameters are set as $\alpha = 1,\ \beta = 5,\ \gamma = 1$ and $\delta = 5$ is considered. 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. This case is denoted as Case 6. For completeness of the presentation of the experimental results, the results for Case 6 are added to each figure.

From here, the experimental results for each task set category are presented. 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.

3.5.1 Small Task Set

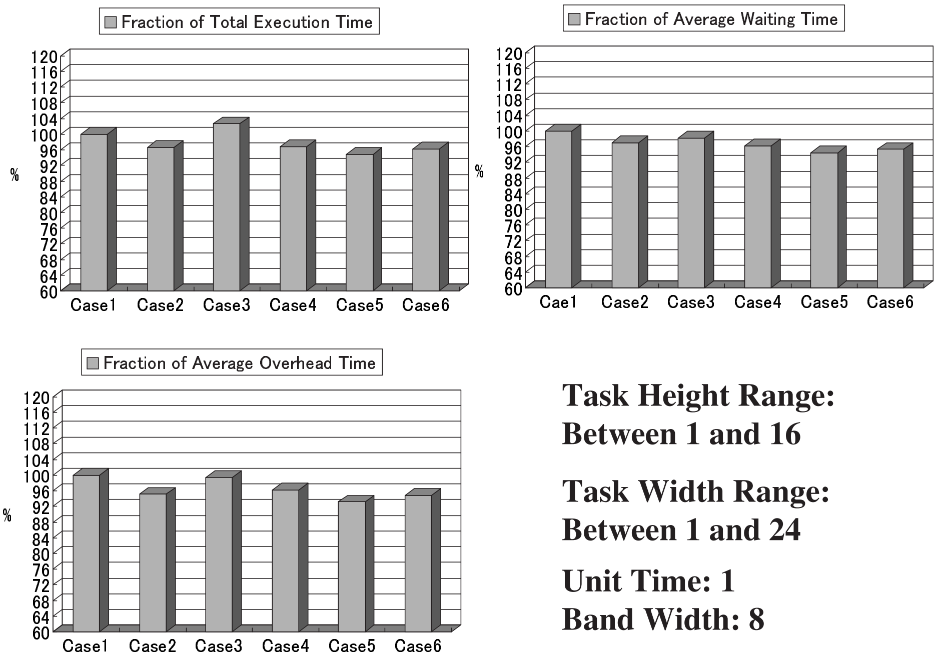
The tasks in a small task set are of height between 1 and 16 and width between 1 and 24. Note that the largest values in task height and width, respectively, are a quarter of those for the FPGA. The experimental results are shown in Figure 3.15. The data for Cases 2 and 4 clearly indicated performance improvements by their corresponding strategies, *I/O Oriented* and *Fragmentation Aware*. Therefore the parameter values for Case 5 are set 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, a value of 30 is assigned 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 $\beta = 40$ for Case 5 is set. It should be noted that this case further improved the results.

3.5.2 Medium Task Set

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. Figure 3.16 depicts the experimental results.

By comparing the results for Case 4 in this and the previous figures, that *Fragmentation Aware* has less impact in the medium task set than in the small task set is observed. 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, the parameters of Case 5 are set as $\alpha = 1,\ \beta = 10,\ \gamma = 0$, and $\delta = 1$.



Task Height Range:
Between 1 and 16

Task Width Range:
Between 1 and 24

Unit Time: 1
Band Width: 8

Figure 3.15. Performance summary for small task set

3.5.3 Large Task Set

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 experimental results are provided in Figure 3.17.

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, the values of parameters of Case 5 are set as $\alpha = 1,\ \beta = 100,\ \gamma = 10$, and $\delta = 5$.

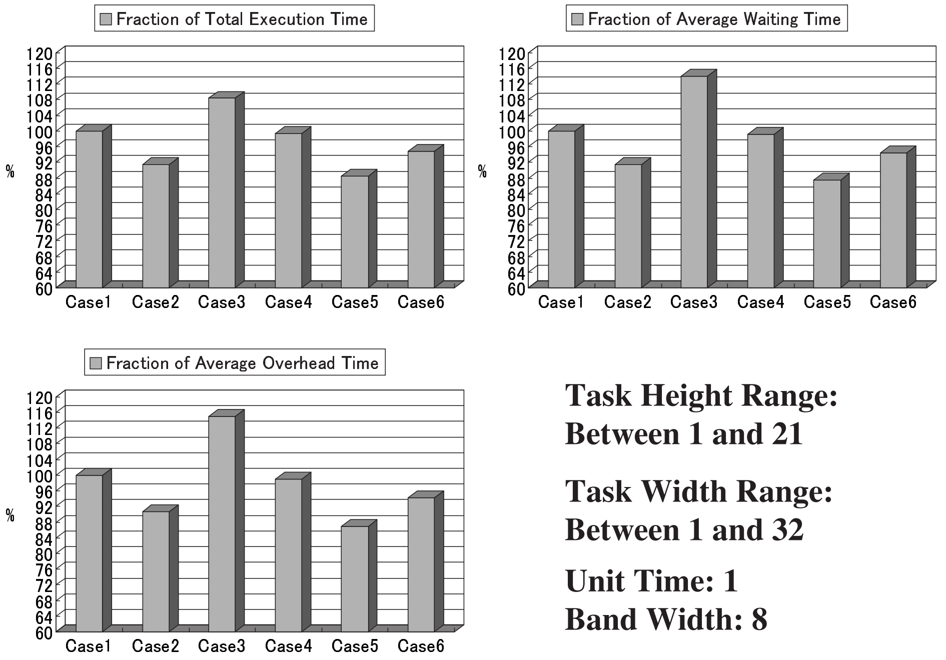


Figure 3.16. Performance summary for medium task set

3.5.4 Performance Comparisons with Different Unit Time Values

In order to observe the effect of I/O communications on the execution of the tasks, Additional experiments are conducted for the small task set with different values set for unit time t_{unit}. The parameter values of Case 5 are used and the three variations in value of 0.1, 1 and 10 for t_{unit} are considered. The results are shown in Figure 3.18.

The first case of $t_{unit} = 0.1$ means a low communication cost scenario. In this case, not much performance impact is observed. On the other hand, for the third case of $t_{unit} = 10$, the method shows a remarkable performance improvement over the conventional method. This would lead us to conclude that the new model and new online task placement algorithm play an important role when high I/O communication costs are incurred.

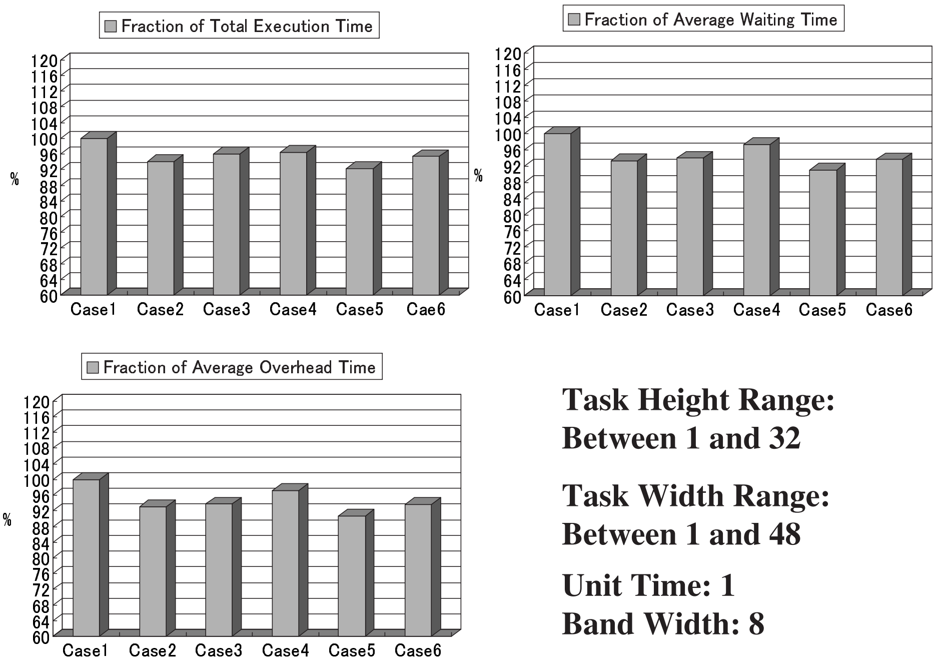
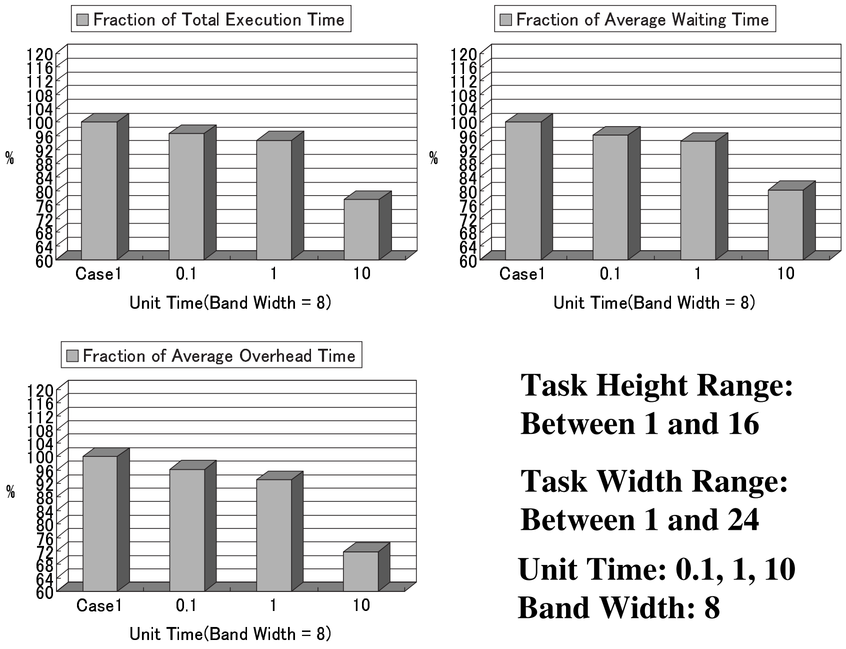


Figure 3.17. Performance summary for large task set

3.5.5 Overhead of the Placement Engine

As mentioned earlier, the use of an online task placement algorithm should not incur a large overhead in order to achieve an overall efficiency in the entire process of concurrent task execution and task placement. In order to measure the magnitude of overhead, the algorithm is run on the three different task sets with the Case 5 parameter values again. The task execution times range from 0.1 to 38.4 seconds for the small task set, from 0.1 to 67.2 seconds for the medium task set, and from 0.1 to 153.6 seconds for the large task set.

The total execution time was measured from the arrival time of the first task to the finish time of the task completed last. As for the overhead, first the times required for the execution of the placement algorithm including updating operations of all data structures used is summed up. Then the sum is divided by 500 to derive the average overhead time per task. Table 3.1 summarizes the total



Task Height Range: Between 1 and 16

Task Width Range: Between 1 and 24

Unit Time: 0.1, 1, 10
Band Width: 8

Figure 3.18. Performance for different unit time values

execution time and placement overhead time thus obtained for each of the three task sets.

From the table, that the placement algorithm ran very fast and thus the overhead was very small is observed. Note that after each task is placed, the algorithm updates the data stored in the data structures such as the area and hash Matrices. These updating operations are being performed while the tasks that have been placed in the FPGA are being executed. Therefore, the times involved in these operations are in fact not part of the overhead. Therefore, the real overhead time is much smaller than those shown in the table. Thus that the overhead time of the placement engine is negligible is safely concluded.

Task Set	Total Execution Time	Placement Overhead Per Task
Small	3985.1	8.2543×10^{-2}
Medium	4774.3	8.5166×10^{-2}
Large	9103.9	11.638×10^{-2}

Table 3.1. Total execution time and placement overhead

3.6. Conclusions

In this chapter, a new model for online FPGA placement was introduced. Unlike the conventional model, the model considers the effect of communications between the tasks and the I/O elements on the periphery of a partially reconfigurable FPGA. Two fitting strategies were proposed for task placement in the model. An online task placement algorithm was developed. It selects an empty area for each task using a properly weighted combination of these two and two other previously used fitting strategies. By simulations performance improvements of the algorithm over the conventional method were shown. That the overhead time incurred by running this algorithm was negligible was further shown.

The model assumes an unlimited amount of resources for communication channels. There is a plan to expand the work for partially reconfigurable FPGAs with a limited number of communication channels. In such a system, it would make more sense to place tasks in such a way that certain portions of rows/columns of CLBs between neighboring tasks are kept unused. Therefore a new placement method that incorporates this intentional insertion of unused space to fully accommodate I/O communications of the tasks is needed.

Chapter 4

A Placement and Routing Algorithm for a Reconfigurable 1-bit Processor Array

4.1. Introduction

Recent advances in reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices [6] has promoted their use in many computing systems to improve their performance. In particular, FPGAs are widely used due to the flexibility of reconfiguring their functionality as well as their enhanced development environment.

However, some flaws of FPGAs limit the fields to which they can be applied. First, since wiring areas are dominant in chip areas of FPGAs, causing their logic density to decrease, an implementation of a target application might not fit into their logic areas. Second, the clock frequency of FPGAs is relatively low due to their fine-grained structure. As a result, desired performance cannot be obtained.

To overcome these drawbacks, some devices have been proposed. For example, Ohta et al. proposed an FPGA architecture with bit-serial pipeline data paths [34]. However, the base of their architecture is still similar to conventional FPGAs. Among devices proposed, a reconfigurable 1-bit processor array (1-bit RPA) [33] is considered to be one of the most significant results. A 1-bit RPA has the structure of a bit-serial data path, but its features are reduced wiring areas, flexible routability, high logic density, and high clock frequency.

A well developed design environment is an essential factor for a new type of architecture, since it is impossible for a large-scale design to be implemented by hand. In this field, commercial FPGAs have arisen in an advanced development environment, and many effective methods have been proposed such as versatile place and route [5]. On the other hand, there has been no design environment for a 1-bit RPA yet proposed. However, existing methods of FPGAs cannot simply be applied to a 1-bit RPA since the structures of 1-bit RPAs and FPGAs are substantially different. In the architecture, a processor element can be used to bridge wires for more flexible mapping, making placement and routing integrated and complicated. Therefore, an efficient way to map target applications to the architecture is needed.

In this chapter, an efficient and effective placement and routing algorithm for a reconfigurable 1-bit processor array is proposed. A 1-bit RPA features a unique wiring structure that makes possible flexible mapping of applications. A proposed placement and routing algorithm achieves compact implementation through various optimization steps by taking advantage of the 1-bit RPA's characteristic routing structure.

In the next section, a brief explanation of the architecture of a reconfigurable 1-bit RPA is described. In Section 4.3, the placement and routing algorithm for a reconfigurable 1-bit PA is explained. Then, experimental results of an implementation of the proposed algorithm are presented in Section 4.4. Section 4.5 summerizes this chapter and describes some future work.

4.2. Architecture of a Reconfigurable 1-bit Processor Array

In this section, an explanation of the 1-bit RPA's architecture is presented.

4.2.1 Overview of a Reconfigurable 1-bit Processor Array

As mentioned earlier, the 1-bit RPA has the features of a bit-serial data path and a unique wiring network. The wiring areas have been reduced compared

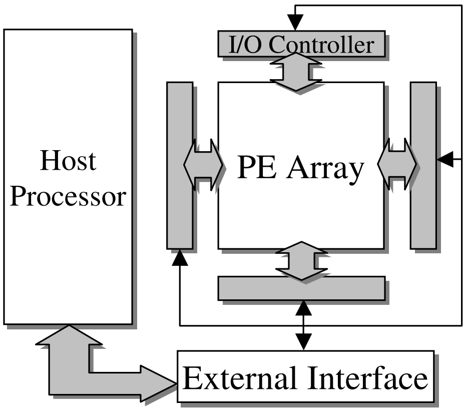


Figure 4.1. Block diagram of a 1-bit RPA

to conventional FPGAs due to its structure, and its performance is superior, demonstrated by an example application such as a discrete cosine transform [33]. However, owing to these features, the mapping process of a 1-bit RPA is different from that of conventional methods.

Figure 4.1 shows a block diagram of a 1-bit RPA. In the figure, a processor element array is the main component of the architecture and I/O controllers are functional units for controlling input and output data from and to external components. The processor element array is composed of processor elements (PEs) and I/O elements (IOEs) that are primal processing units and controllers to feed and receive data to and from PEs, respectively. PEs are capable of basic arithmetic operations such as addition, subtraction, and shift. IOEs are arranged at the periphery of PEs.

4.2.2 Wiring Structure of a Reconfigurable 1-bit Processor Array

Next, the details of the 1-bit RPA's interconnection network are described. The architecture's wiring structure has two types of wiring resources (short wires and long wires), and PEs can be used to bridge wires. Short wires are used to transfer data between neighboring PEs, with Figure 4.2 showing the structure of short

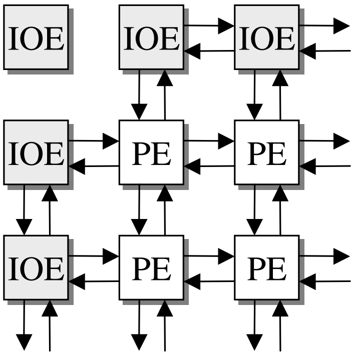


Figure 4.2. Structure of short wires

wires. A PE has four input and output short wires, respectively. Long wires are used to transfer data among distant PEs. The structure of long wires is depicted in Figure 4.3. The architecture also includes two static parameters, *distance* and *step*, which determine the configuration of long wires and are set before fabrication. Distance denotes the farthest PE that a PE can access through a long wire, and step represents intervals of long wires. In Figure 4.3, the left-hand figure depicts the case where distance is 3 and step is 1. This means a PE can access PEs within a distance of 3 and wires with a distance of 3 are arranged between every side of PEs. Here, only wires in rows are shown for clarity. The figure shows that PE_a has access to PE_b and PE_c using a long wire. The right side of the figure shows an example of the case where distance is 2 and step is 2. In the right-hand figure, PE_a uses PE_d as a bridge to connect to PE_c due to the shortness of long wires. As these examples indicate, the parameters of long wires affect the mapping flexibility.

Before proceeding further, one more feature of the architecture must be described. Because a 1-bit RPA employs the bit-serial data path structure, delays may occur due to certain operations such as multiplication or shift. Using a PE for bridging wires also leads to a delay. Here, a delay means extra clock cycles and a PE can support up to delays of 32 clock cycles. Therefore, it is necessary to adjust the data timings. The 1-bit RPA has two ways to deal with adjustment of data timings. One is a way to insert a PE for bridging wires to delay data

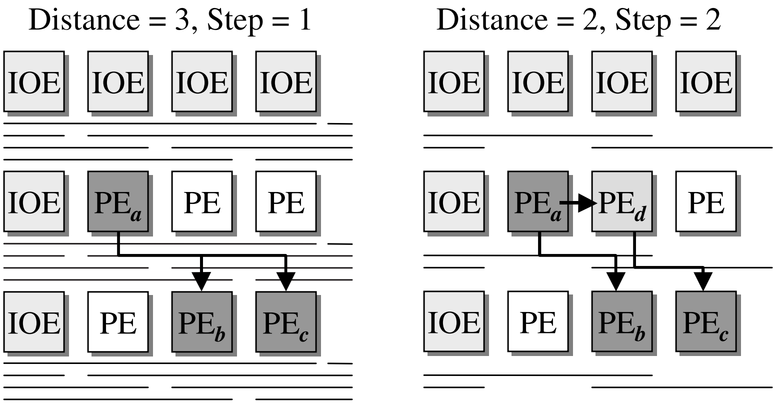


Figure 4.3. Structure of a long wire

transfers. A demonstrative example of this is depicted in Figure 4.4. In the figure, PE_g can have direct access to PE_h using a long wire located immediately above PE_h and PE_i. PE_h's input from PE_e has a delay through PE_f. Therefore, it is necessary for PE_h's input from PE_g to have the same number of delays and, here, it has a delay through PE_i. However, this method leads to more complicated placement and routing and impedes the placement and routing output to shrink. The other way to handle this problem is a method in which a PE has input and output buffers for adjusting data timings. This method leads to an increase of the PE area. However, in terms of placement and routing quality, this method is better since it does not increase the number of PEs used. Therefore, in this dissertation, the way to use input and output buffers is employed and the number of buffers needed is dealt with architectural parameter as is the case with step and distance.

4.3. Placement and Routing Algorithm for a Reconfigurable 1-bit Processor Array

In this section the processes of the placement and routing algorithm is explained in detail. This part starts with an overview of the mapping processes.

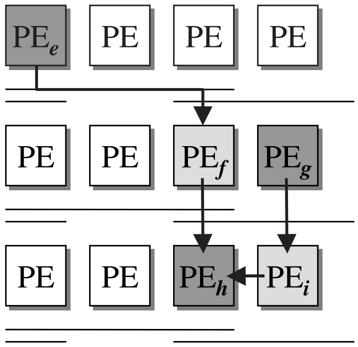


Figure 4.4. Insertion of PEs for delay

4.3.1 Overview of Mapping Algorithm

Figure 4.5 shows the whole process of the proposed placement and routing algorithm. The process of the mapping algorithm comprises three steps: initial placement, initial routing, and iterative optimizations. A control data flow graph (CDFG) is employed as an input of the algorithm. That input CDFGs include delay elements to adjust the data timings caused by certain operations is assumed. The current algorithm deals with delay elements of CDFGs as a node. An output is placement and routing information for the 1-bit RPA's configuration data. In the strategy, empty PEs that are used to route nodes are purposely inserted. Here, empty PEs mean PEs that are reserved for bridging wires and these PEs are called connection PEs (CPEs). Consequently, in the initial placement, the algorithm does not use these empty PEs to place nodes of CDFGs. The number of empty PEs is dealt with as a parameter that can be changed according to need. This scheme is one of the techniques of this placement and routing algorithm.

The first step of the algorithm is the initial placement stage. Before proceeding to the placement stage, the number of empty PEs reserved for CPEs is determined in the initial setting stage and these empty PEs are placed between each node. Initial placement with no empty cells might lead to unsuccessful mapping outputs; therefore, the possibility of producing successful mapping results is raised by placing these empty PEs between each node.

In the placement stage, there are following two strategies.

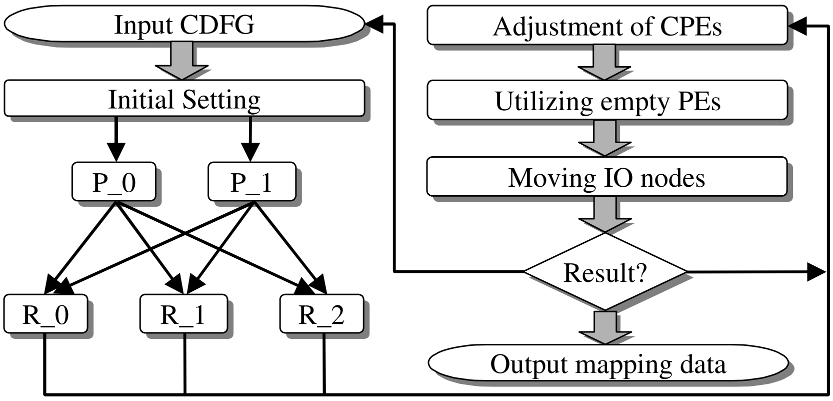


Figure 4.5. Flow of mapping algorithm

- P_0, No Backtrack
- P_1, Square

In the first strategy, (P_0), nodes of a CDFG are arranged in the form in which no backward flows are structured and this is the same form as an output of the task graph for free (TGFF) [14]. The first strategy, (P_1), is the placement whereby nodes are placed in the square-shaped form. Both strategies employ the scheme in which each node is swapped to minimize the average Manhattan distance (MD) between two nodes. However, in P_0 strategy, each node is swapped only in a lateral direction. In P_1 strategy, nodes are swapped in a lateral and longitudinal direction. By iterating swapping of nodes, both methods aim to minimize total MD between each node.

The next step is the initial routing stage. In this stage, there are following three strategies.

- R_0, Degree
- R_1, Manhattan distance
- R_2, Degree + Manhattan distance

In the first strategy (R_0), the routing begins with the node that has the largest input and output degrees. The routing of the second one (R_1) is done in descending order from the node that has the longest MD. The last one, (R_2), is the combined method of R_0 and R_1. The routing of R_2 starts with the node whose sum of degree and MD is the largest.

In every routing step, the algorithm searches for the longest long wires for wiring two connected nodes and the selection of directions is decided at random when it routes two connected nodes. If there are no long wires available, routing engine searches for long wires at the other direction. If the long wires of both directions are not available, routing algorithm tries to use short wires and neighboring empty nodes to bridge wires. If none of the long wires, short wires and empty nodes are available, the algorithm stops at that point and start routing from the initial state by changing pseudorandom seeds. In the initial routing stage, empty PEs that were placed between nodes in the previous stage are utilized as CPEs, which makes possible various routings among nodes.

In the initial placement and routing stage, the algorithm employs one of the above-described strategies, and the strategy selection is performed in the initial setting stage before entering placement and routing processes. Since strategies of these stages are naive ones, the placement quality needs to be improved by the following optimization stages described in the next section.

4.3.2 Optimization Schemes of Placement and Routing Engine

Here, optimization schemes of the mapping algorithm are presented. They are composed of three steps. The following cost function is evaluated in each step.

$$Cost \;=\; \alpha \cdot \#CPEs \;+\; \beta \cdot \sum_i center(P_i) \tag{4.1}$$

The first term represents the total number of CPEs, and the second represents the sum of $node_i$'s Manhattan Distance to the central point; α and β are user-defined parameters. The central point is the center-most position of the most congested area, which is determined before entering the optimization steps. If this cost function becomes smaller at each optimization step, the algorithm takes the result of the optimization as a new state of the placement and routing. Iterating the following three optimization steps an arbitrary number of times, the placement and routing engine attempts to improve the quality of the mapping result.

Optimization 1: *Adjustment of CPEs*
In this optimization step, the algorithm attempts to perform swapping between each node and CPEs that are connected to the node. If it is possible to re-route the nodes and the cost function decreases, swapping of nodes is performed. This step is illustrated using an example. In the following figures, the architectures on the left side show the unoptimized state while those on the right side show the optimized state. Dark gray, light gray, and white squares respectively denote occupied, connection, and empty PEs. That the central point exists in the lower part of each figure is assumed. For illustrative purposes, only the intended figures are shown.

Let us focus on *node* 23 and the CPE that is placed between *node* 23 and *node* 25 in the left side of Figure 4.6. If *node* 23 is moved to where the CPE is placed and can be re-routed there, the cost function is evaluated. If the cost becomes smaller, swapping between the node and the CPE is performed. Although the number of CPEs increases in this example, *node* 23 comes closer to the central point. Therefore, it depends on the parameters whether the result of this optimization is used.

Optimization 2: *Utilizing empty PEs*
The second optimization is an approach to utilizing empty PEs. In this step, the algorithm investigates whether each node can be moved to empty PEs. If it is possible to displace a node and, as a result, the cost decreases, the swapping between a node and an empty node is performed. In Figure 4.7, *node* 1 can be moved to the bottom-left PEs, leading to *node* 1's proximity to the central point.

Optimization 3: *Moving input and output nodes*
Moving input and output nodes is the last optimization step. In this optimization step, the algorithm does swaps between input and output nodes and empty IOEs. If the swapping can be done and the input and output nodes are placed at an inner place, the result of this optimization step is taken as a new status of the placement and routing. When all PEs at the outermost lines of the PE area become empty, the line is changed to IO node area and is used as the swapping

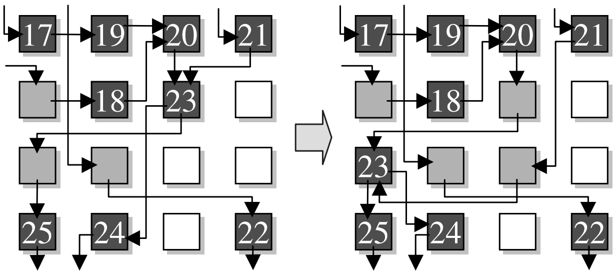


Figure 4.6. Adjustment of CPEs

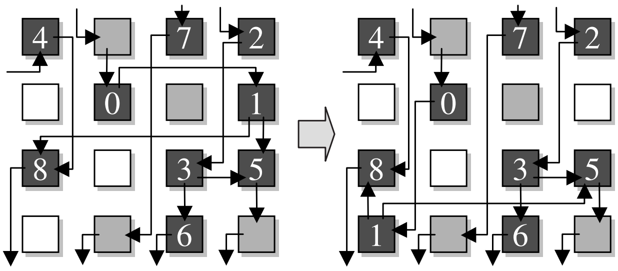


Figure 4.7. Utilizing empty PEs

point of already placed input and output nodes. This optimization step aims to shrink the total area of the placement and routing output by moving IO nodes to inner spaces.

4.4. Experimental Results

In this section, experimental results of the placement and routing engine are shown.

4.4.1 Experimental Results of Initial Placement and Routing

Here, experimental results of initial placement and routing are described. Table 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7 show the results of initial placement and routing using CDFG example 1 of Figure 4.8 with placemat and routing strategies and parameter setting changed. The CDFG example 1 has 40 nodes, in-degree limits are 3, and out-degree limits are 2. Table 4.1 shows the maximal number of long wires between neighboring two nodes with various parameter setting. The maximal number of long wires is obtained from Equation 4.2.

The number of the successful placement and routing is shown in Table 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7 when a hundred times of placement and routing is executed with pseudorandom seeds varied. In each table, $-$ represents none of the placement and routing executions are successful. As for the successful placement and routing, right-hand values of each table represent the number of the successful placement and routing and left-hand values represent the number of average CPEs used to routing nodes. In this initial placement and routing experiment, an empty node is inserted between neighboring two nodes at the initial placement stage and this is shown in each table as $offset = 1$.

From the experimental results of 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7, when the parameter, step, is 1, many of the placement and routing are successfully performed. On the other hand, when step is 3, all of the placement and routing failed. A 1-bit RPA with step 3 has few long wires as shown in Figure 4.1. Therefore, resource conflicts could occur and this leads to the unsuccessful placement and routing. As a result, the parameter, step, has a more dominant influence on the success and failure of the placement and routing, although there is a certain influence of the parameter, distance.

Next, the effect of placement and routing strategies is focused. From the experimental results of 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7, P_1 placement leads to more successful placement and routing compared to P_0 placement. It can be considered that in the P_0 placement relatively long MDs between two connected nodes still remain after swapping nodes due to no backtracking strategy and the long distance between two nodes could lead to unsuccessful mapping results. On the contrary, the differences between routing strategies do not have a significant

Step \ Distance	1	2	3	4	5
1	2	3	4	5	6
2	1	2	2	3	3
3	1	1	2	2	2

Table 4.1. Maximal number of various parameter setting

$$L_{max} = \lceil \frac{distance + 1}{step} \rceil \tag{4.2}$$

impact on the number of successful mapping.

Table 4.8 shows the effect of the number of empty nodes initially inserted between neighboring two nodes. In the table, $ePEs$ represents the number of empty nodes inserted at the initial placement stage. From the table, it is clear that the number of the successful placement and routing increases when two empty nodes are inserted between each node. Especially, the successful initial mapping with $step = 1$ and $distance = 2$ appeared only using two empty nodes ($offset = 2$). However, the cases with $offset = 2$ need much more CPEs than the cases with $offset = 1$ and this should lead to the difficulty of optimization stages.

4.4.2 Experimental Results of Optimization of Placement and Routing

Next, the effectiveness of optimization steps is shown. Here, CDFG example 1, CDFG example 2, and CDFG example 3 are used to evaluate the optimization steps. These CDFGs are obtained from TGFF's output and shown in Figure 4.8, 4.9, and 4.10, respectively. The property of each CDFG is shown in Table 4.9. The parameters of distance and step are set to 5 and 1, respectively. The placement strategy is P_1 and the routing strategy is R_2.

$offset = 1,\ P\_0(No\ Backtrack),\ R\_0(Degree)$								
Distance Step	1	2	3		4		5	
1	–	–	77.0	1	60.0	44	50.3	77
2	–	–	–		–		–	
3	–	–	–		–		–	

Table 4.2. Experimental result of P_0 placement and R_0 routing

$offset = 1,\ P\_0(No\ Backtrack),\ R\_1(MD)$								
Distance Step	1	2	3		4		5	
1	–	–	72.0	2	57.8	49	49.4	70
2	–	–	–		–		–	
3	–	–	–		–		–	

Table 4.3. Experimental result of P_0 placement and R_1 routing

Table 4.10, 4.11, and 4.12 show the experimental results of ten optimized mapping outputs of each CDFG. In this experiment, five times iterations of optimization steps are applied to the results of the successful initial placement and routing of each CDFG. Table 4.13 and 4.14 also show the summary of the experimental results in which each value shows the averages over the values of Table 4.10, 4.11, and 4.12. Table 4.15 shows the rate of reduction between the initial state and the optimized state.

The averages of maximal delay are described in Table 4.13. These values of the table represent the number of CPEs for bridging wires to route connected nodes. As previously mentioned, using these CPEs causes the data transfer to delay. Therefore, input and output buffers of PEs are used to adjust data timings. From the experimental result, in the case of CDFG 1 and the case of CDFG 2 and 3, three and four buffers are needed respectively to deal with the data timing adjustment. The number of nodes of CDFG 2 and 3 is larger than that of CDFG

$offset = 1$, $P\_0(No\ Backtrack)$, $R\_2(MD + Degree)$								
Step \ Distance	1	2	3		4		5	
1	–	–	74.3	3	58.9	40	49.9	69
2	–	–	–		–		62.0	2
3	–	–	–		–		–	

Table 4.4. Experimental result of P_0 placement and R_2 routing

$offset = 1$, $P\_1(Square)$, $R\_0(Degree)$								
Step \ Distance	1	2	3		4		5	
1	–	–	53.6	27	42.5	83	34.4	93
2	–	–	–		–		42.0	3
3	–	–	–		–		–	

Table 4.5. Experimental result of P_1 placement and R_0 routing

1. Therefore, the CDFGs need more CPEs to route nodes and this leads to more input and output buffers.

In the result of CDFG 1 of Figure 4.15, the average size of the mapping output has shrunk to 51.3% from that of the initial state, and the average number of CPEs has decreased to 34.3% of the initial state. Also, in the CDFG 2 result, the average size of the placement output has become 62.1% from that of the initial state, and the average number of CPEs has decreased to 52.2% of the initial state. From the results of experiment of CDFG 3, the average area of its placement has shrunk to 74.9% from that of the initial state, and the average number of CPEs has decreased to 67.0% of the initial state. From these experimental results, it can be observed that the output of smaller CDFG is better than the other outputs. It can be considered that the parameter, distance, is large enough for the placement and routing of CDFG 1. On the contrary, for CDFG 2 and 3, distance of 5 is not enough to connect their nodes since the MD between each node of CDFG 2

$offset = 1$, $P\_1(Square)$, $R\_1(MD)$								
Distance Step	1	2	3		4		5	
1	–	–	53.3	38	41.6	80	34.3	92
2	–	–	–		46.7	3	48.8	5
3	–	–	–		–		–	

Table 4.6. Experimental result of P_1 placement and R_1 routing

$offset = 1$, $P\_1(Square)$, $R\_2(MD + Degree)$								
Distance Step	1	2	3		4		5	
1	–	–	52.7	30	42.1	74	34.2	93
2	–	–	–		50.0	2	50.0	1
3	–	–	–		–		–	

Table 4.7. Experimental result of P_1 placement and R_2 routing

and CDFG 3 is more distant than that of CDFG 1. Therefore, long wires with larger distance are needed to optimize the results of the placement and routing of CDFG 2 and 3. However, long wires with too large distance leads to the increase of the area of a PE since PE's multiplexers to select many long wires become large. As a result, the balance between the size of a CDFG and the length of a long wire is considered to be more important.

4.5. Conclusions and Future Work

In this chapter, an efficient and effective placement and routing algorithm for a reconfigurable 1-bit processor array was proposed. Since the existing placement and routing methods cannot simply be utilized due to the 1-bit RPA's unique architecture, a dedicated and effective way to map target applications automat-

$P\_1(Square)$, $R\_2(MD + Degree)$										
Distance Step	$ePEs$	1	2		3		4		5	
1	1	–	–	–	52.7	30	42.1	74	34.2	93
	2	–	178	6	114.5	74	87.3	97	70.7	99
2	1	–	–		–		50.0	2	50.0	1
	2	–	–		–		106.1	18	86.4	53
3	1	–	–		–		–		–	
	2	–	–		–		–		–	

Table 4.8. Comparison between $offset = 1$ and $offset = 2$

	$Number\ of\ Nodes$	$Number\ of\ Arcs$
$CDFG$ 1	41	53
$CDFG$ 2	56	72
$CDFG$ 3	70	90

Table 4.9. Property of each CDFG

ically is needed. In the mapping algorithm, initial placement and routing stages employ a naive method, but the proposed algorithm attempts to achieve compact mapping of applications through various iterative optimization steps.

As future work, more detailed experiments using parameters of larger distance should be conducted. Through the experiments, attempts will be made to improve the placement and routing engine and to derive the optimal parameter setting for wide-ranging applications.

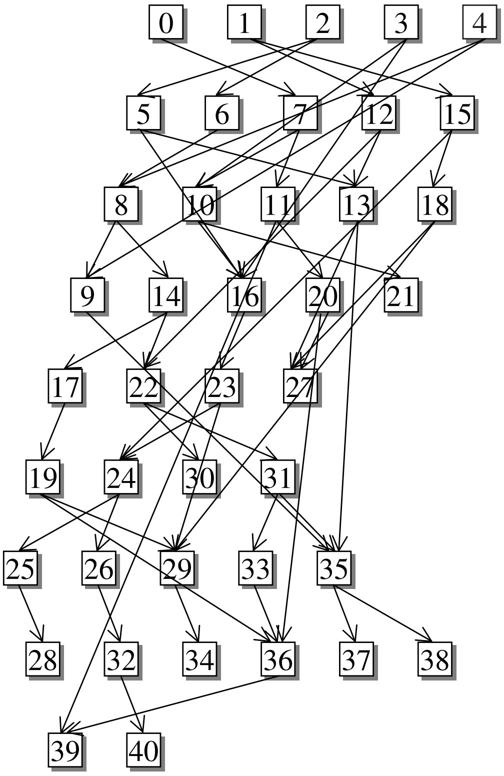


Figure 4.8. CDFG example 1

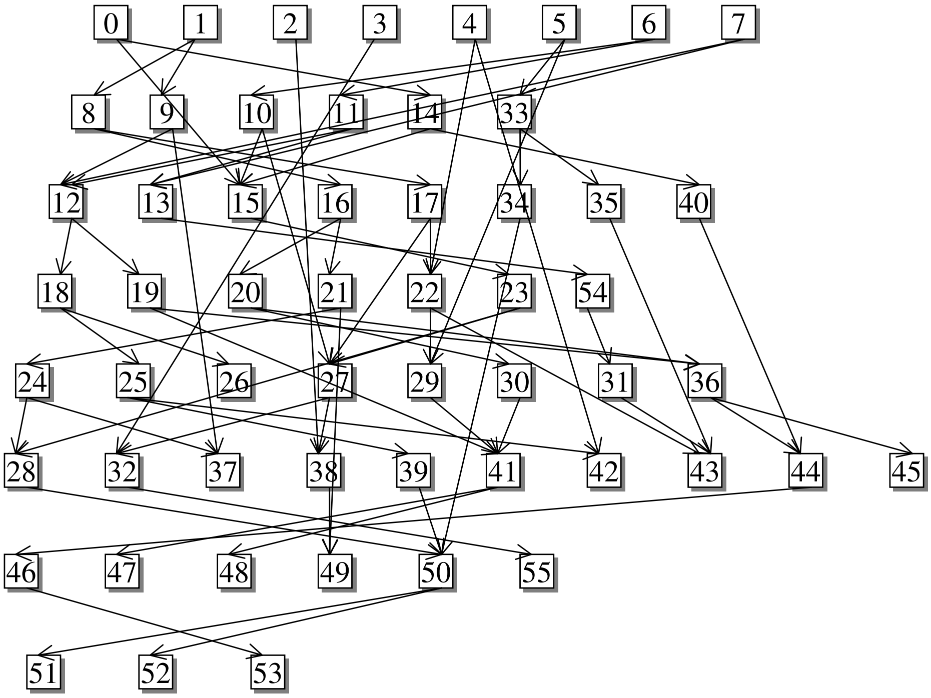


Figure 4.9. CDFG example 2

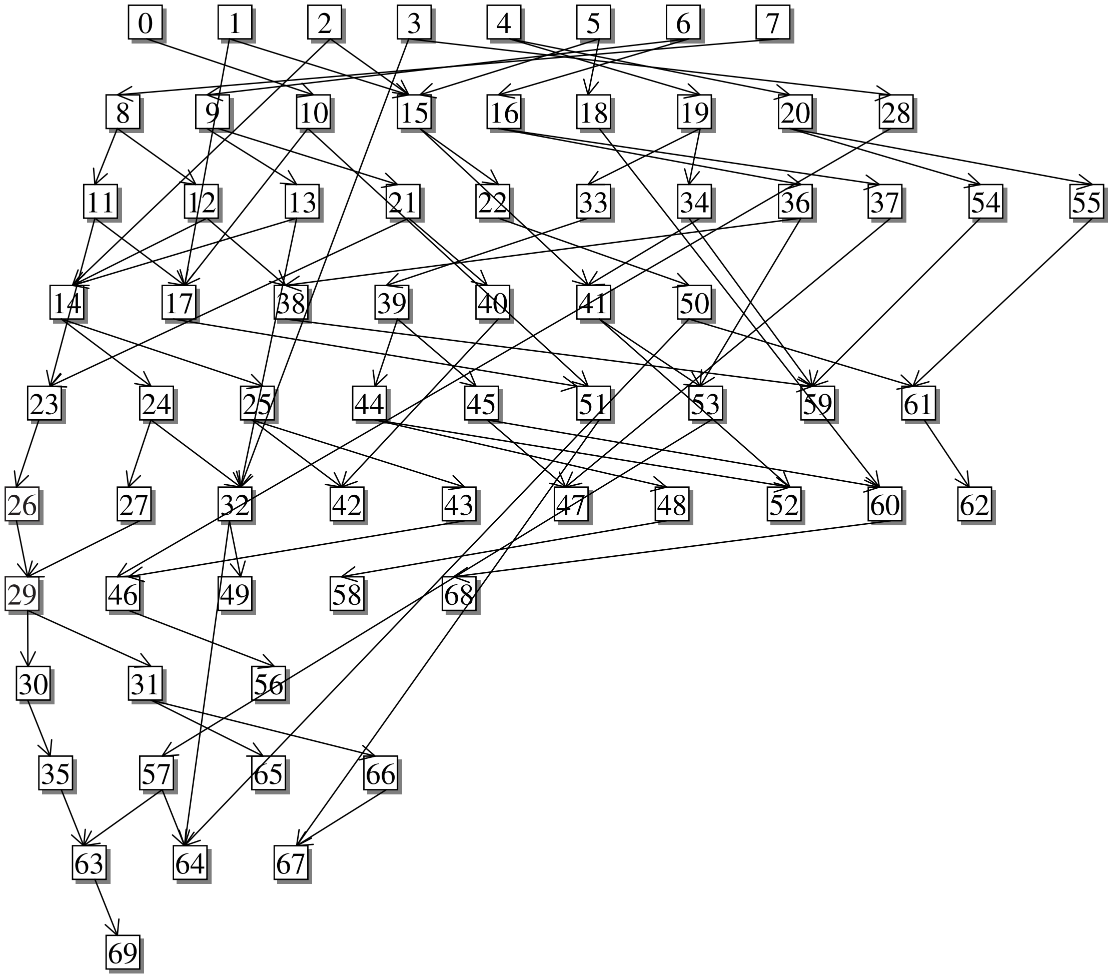


Figure 4.10. CDFG example 3

	$CPE(Initial)$	$CPE(Optimized)$	$Area$	$Max\ Delay$
1	40	11	12×11	2
2	33	20	11×10	3
3	28	9	10×11	2
4	38	6	10×8	1
5	37	15	11×11	2
6	32	13	10×11	4
7	35	15	11×12	3
8	40	9	13×11	2
9	38	7	10×9	1
10	32	16	12×11	3

Table 4.10. Experimental results of optimization of CDFG 1

	$CPE(Initial)$	$CPE(Optimized)$	$Area$	$Max\ Delay$
1	57	27	14×14	3
2	56	20	15×14	2
3	50	27	12×12	2
4	54	38	13×14	3
5	56	24	12×13	4
6	55	23	11×13	4
7	61	31	14×14	3
8	53	32	12×14	5
9	56	40	14×14	4
10	60	29	14×15	4

Table 4.11. Experimental results of optimization of CDFG 2

	$CPE(Initial)$	$CPE(Optimized)$	$Area$	$Max\ Delay$
1	77	68	18 × 17	3
2	76	38	16 × 14	4
3	79	76	18 × 17	4
4	75	38	15 × 16	4
5	72	25	16 × 15	2
6	70	45	18 × 15	3
7	75	72	18 × 18	5
8	77	37	16 × 15	2
9	80	72	18 × 17	4
10	79	38	16 × 16	3

Table 4.12. Experimental results of optimization of CDFG 3

	$CPE(Initial)$	$CPE(Optimized)$	$Maximal\ Delay$
$CDFG$ 1	35.3	12.1	2.3
$CDFG$ 2	55.8	29.1	3.4
$CDFG$ 3	76	50.9	3.4

Table 4.13. Summary of optimized placement and routing (CPE)

	$Area(Initial)$	$Area(Optimized)$
$CDFG$ 1	15 × 15	11.0 × 10.5
$CDFG$ 2	17 × 17	13.1 × 13.7
$CDFG$ 3	19 × 19	16.9 × 16.0

Table 4.14. Summary of optimized placement and routing (Area)

	$Reduction(Area)$	$Reduction(CPE)$
$CDFG$ 1	51.3%	34.3%
$CDFG$ 2	62.1%	52.2%
$CDFG$ 3	74.9%	67.0%

Table 4.15. Reduction of area and CPEs of optimized placement and routing

Chapter 5

Conclusion

Reconfigurable computing is now one of the promising techniques to boost up the total system performance. Therefore, further utilization methods of the systems are needed. In this dissertation, some design environments for reconfigurable computing systems have been proposed.

In Chapter 2, the method of utilizing the hybrid system is presented. In the system, a general-purpose processor and reconfigurable logics are combined. In such hybrid systems, the extracted parallelism can be spoiled by the communication bottleneck of the system. A characteristic of Event-Oriented Computing can avoid the bottleneck and, therefore, the applications' parallelism can not be spoiled. An architecture model for Event-Oriented Computing was proposed and the effectiveness of the system was evaluated using an artificial life program. The model achieved a processing time about 3.70 times faster than software implementation in the average case. As future work, the system need to be implemented at a greater depth to make the effectiveness of the solution more clear and it is necessary for further applications to be investigated.

In Chapter 3, the effective online task placement algorithm for a partially reconfigurable FPGA was proposed. The task placement method employs task's I/O routing information for efficient processing of the incoming tasks. Fitting strategies that were proposed and previously proposed strategies are combined and properly weighted to make an effective placement of the tasks. Experimental results have demonstrated the effectiveness of the online task placement engine compared to the conventional method. An unlimited amount of resources for communication channels are assumed in the model. Therefore, as future work, the algorithm needs to be improved for partially reconfigurable FPGAs with limited number of communication channels.

In Chapter 4, a placement and routing algorithm for a reconfigurable 1-bit processor array was proposed. The unique wiring structure of a 1-bit RPA avoids the simple use of existing placement and routing method. So, an innovative mapping method was developed. The placement and routing engine places empty PEs between nodes. In the optimization stage, the algorithm attempts to achieve compact placement by utilizing the empty PEs. In the small CDFG example, experimental results showed that the size of the mapping output shrunk to 51.3% from that of the initial state, and the number of CPEs decreased to 34.2% of the initial state. As future work, detailed and wide experiments using parameters of larger distance need to be conducted to show that the proposed placement and routing algorithm is an effective method to larger CDFGs.

Acknowledgement

I would like to express my genuine appreciation to Professor Yasuhiko Nakashima of Nara Institute of Science and Technology for his helpful suggestions, accurate criticisms, and invaluable support for this research.

I would like to show my appreciation to Professor Hideo Fujiwara of Nara Institute of Science and Technology for his invaluable comments and careful review concerning this dissertation.

I am heartily grateful to Associate Professor Shigeru Yamashita of Nara Institute of Science and Technology for his continuous guidance, patient supports, and pertinent remarks.

I wish to appreciate Emeritus Professor Katsumasa Watanabe of Nara Institute of Science and Technology for his helpful suggestions, generous supports, and accurate criticisms.

I would like to express my deep gratitude to Professor Kazuo Nakajima of University of Maryland, College Park for his helpful suggestions, accurate remarks, and invaluable supports for my research activity in the United States.

I would like to thank Assistant Professor Masaki Nakanishi of Nara Institute of Science and Technology for his constructive discussions, helpful advice, and continuous encouragement.

I am obliged to all of my friends and colleagues of Nakashima Laboratory for their discussions and helpful comments.

Lastly, I thank my family for their patience, continuous supports, and continuous encouragement.

References

[1] Altera Corporation, “Quartus II Software,” http://www.altera.com/products/software/products/quartus2/qts-index.html

[2] H. Amano, “Recent trends on Reconfigurable/Dynamically Reconfigurable Systems,” In Proc. of IEICE Technical Report of SR2005-5, Vol. 105, No. 36, May 2005, pp. 31–36, (In Japanese).

[3] H. Amano, “A Survey on Dynamically Reconfigurable Processors,” In Proc. of IEICE Technical Report of ICD2003-130, Vol. 103, No. 382, Oct. 2003, pp. 47–52, (In Japanese).

[4] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast Template Placement for Reconfigurable Computing Systems,” IEEE Design and Test of Computers, Vol. 17, No. 1, pp. 68–83, 2000.

[5] V. Betz and J. Rose, “VPR: A New Packing, Placement and Routing Tool for FPGA Research,” In Proc. of the 7th International Workshop on Field-Programmable Logic and Applications, London, UK, 1997, pp. 213–222.

[6] S. Brown and J. Rose, “Architecture of FPGAs and CPLDs: A Tutorial,” IEEE Design and Test of Computers, Vol. 13, No. 2, pp. 42–57, 1996.

[7] D. A. Buell and K. L. Pocek, “Custom Computing Machines: An Introduction,” The Journal of Supercomputing, Vol. 9, No. 3, pp. 219–230, 1995.

[8] T. Callahan, J. R. Hauser and J. Wawrzynek, “The GARP Architecture and C Compiler,” IEEE Computer, Vol. 33, No. 4, pp. 62–69, Apr. 2000.

[9] K. Compton, Z. Li, J. Cooley, S. Knol and S. Hauck, “Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 10, No. 3, pp. 209–220, 2002.

[10] K. Compton and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software,” ACM Computing Surveys, Vo. 34, No. 2, pp. 171–210, 2002.

[11] A. DeHon, “Reconfigurable Architectures for General-Purpose Computing,” Ph. D. Thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.

[12] A. DeHon, “Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density,” In Proc. of the Fourth Canadian Workshop on Field-Programmable Devices, Toronto, Canada, May, 1996, pp. 47–54.

[13] R. P. Dick and N. K.Jha, “CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems,” In Proc. of International Conference on Computer-Aided Design, San Jose, CA, Nov. 1998, pp. 62–68.

[14] R. P. Dick, D. L. Rhodes, and W. Wolf, “TGFF: task graphs for free,” In Proc. of the 6th international workshop on Hardware/software codesign, Seattle, WA, Mar. 1998, pp. 97–101.

[15] S. Fekete, E. Köhler, and J. Teich, “Optimal FPGA Module Placement with Temporal Precedence Constraints,” In Proc. of Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, Mar. 2001, pp. 658–665.

[16] M. G. Gericota, G. R. Alves, M. L. Silva and J. M. Ferreira, “Run-Time Management of Logic Resources on Reconfigurable Systems,” In Proc. of Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, Mar. 2003, pp. 974–979.

[17] M. Handa and R. Vemuri, “Area Fragmentation in Reconfigurable Operating Systems,” In Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, Jun. 2004, pp. 77–83.

[18] M. Handa and R. Vemuri, “An Integrated Online Scheduling and Placement Methodology,” In Proc. of International Conference on Field Programmable Logic and Application, Leuven, Belgium, Aug./Sept. 2004, pp. 444–453.

[19] M. Handa and R. Vemuri, “A Fast Algorithm for Finding Maximal Empty Rectangles for Dynamic FPGA Placement,” In Proc. of Design, Automation and Test in Europe Conference and Exhibition, Feb. 2004, pp. 744–745.

[20] M. Handa and R. Vemuri, "An Efficient Algorithm for Finding Empty Space for Online FPGA Placement," In Proc. of The 41st Design Automation Conference, San Diego, CA, Jun. 2004, pp. 960–965.

[21] S. Huack, T. W. Fry, M. M. Hosler and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," In Proc. of IEEE Symposium on FPGA-Based Custom Computing Machines, Napa Valley, CA, Apr. 1997, pp. 87–97.

[22] J. R. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," In Proc. of the 5th IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, Apr. 1997, pp. 24–33.

[23] B. Kastrup, J. V. Meerbergen, and K. Nowak, "Seeking (the right) Problems for the Solutions of Reconfigurable Computing," In Proc. of the 9th International Workshop on Field Programmable Logic and Applications, Glasgow, UK, Sep. 1999, pp. 520–525.

[24] R. Laddaga, "Active Software," In Proc. of The First International Workshop on Self-Adaptive Software, IWSAS 2000, Oxford, UK, Apr. 2000, pp. 11–19.

[25] W. H. Mangione-Smith and B. Hutchings, "Reconfigurable Architectures: The Road Ahead," In Proc. of The 4th Reconfigurable Architectures Workshop, Geneva, Switzerland, Apr. 1997, pp. 81–96.

[26] W. H. Mangione-Smith et al., "Seeking Solutions in Configurable Computing," IEEE Computer, Vol. 30, No. 12, pp. 38–43, Dec. 1997.

[27] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," In Proc. of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Monterey, CA, Feb. 1998, pp. 261.

[28] T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," In Proc. of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines, Los Alamitos, CA, Apr. 1998, pp. 2–11.

[29] P. Merino, J. C. Lopez, and M. Jacome, "A Hardware Operating System for Dynamic Reconfiguration of FPGAs," In Proc. of International Workshop

on Field Programmable Gate Arrays, Tallinn, Estonia, Aug./Sept. 1998, pp. 431–435.

[30] M. Motomura, "A Dynamically Reconfigurable Processor Architecture," Microprocessor Forum, San Jose, CA, Oct. 2002.

[31] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi, "Plastic cell architecture: towards reconfigurable computing for general-purpose," In Proc. of FPGAs for Custom Computing Machines, Napa Valley, CA, Apr. 1998, pp. 68–77.

[32] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi, "Plastic cell architecture: A scalable device architecture for general-purpose reconfigurable computing," IEICE Transactions on Electronics, Vol. E81-C, No. 9, pp. 1431-1437. Sep. 1998.

[33] N. Nakai, M. Nakanishi, S. Yamashita, and K. Watanabe, "Reconfigurable 1-Bit Processor Array with Reduced Wiring Area," In Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, Jun. 2005, pp. 225–231.

[34] A. Ohta, T. Isshiki, and H. Kunieda, "New FPGA architecture for bit-serial pipeline datapath," In Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, Napa Valley, CA, Apr. 1998, pp. 58–67.

[35] PCI-SIG,
http://www.pcisig.com/

[36] B. Radunović and V. Milutinović, "A Survey of Reconfigurable Computing Architectures," In Proc. of the 8th International Workshop on Field Programmable Logic and Applications, Tallin, Estonia, Aug./Sep. 1998, pp. 376–385.

[37] R. Razdan and M. D. Smith, "PRISC: Programmable Reduced Instruction Set Computers," Ph. D. Thesis, Harvard University, Division of Applied Science, Cambridge, MA, 1994.

[38] R. Razdan and M. D. Smith, "High-performance Microarchitectures with Hardware programmable Functional Units," In Proc. of the 27th Annual IEEE/ACM International Symposium on Microarchitecture, Nov./Dec. San Jose, CA, 1994, pp. 172–180.

[39] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications," IEEE Transactions on Computers, vol. 49, No. 5, pp. 465–481, 2000.

[40] Y. Shoham, "Agent-Oriented Programming," Artificial Intelligence, Vol. 60, No.1, pp. 51–92, 1993.

[41] T. Sueyoshi and H. Amano ed., "Reconfigurable System," Ohmsha, 2005, (In Japanese).

[42] J. Villasenor and W. H. Mangione-Smith, "Configurable Computing," Scientific American, Vol. 276, No. 6, pp. 54–59 Jun. 1997.

[43] H. Walder, C. Steiger, and M. Platzner, "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing, " In Proc. of International Parallel and Distributed Processing Symposium, Nice, France, Apr. 2003, pp. 178.

[44] K. Watanabe, A. Inoue, M. Tomono, K. Kurakawa, M. Nakanishi, and S. Yamashita, "Assertion Verification Design by Active Function," JSSST Journal on Computer Software, Vol. 22, No. 3, pp.76–91, Jul. 2005, (In Japanese).

[45] G. Weiss ed., "Multiagent systems, A Modern Approach to Distributed Artificial Intelligence," The MIT Press, 1999.

[46] G. Wigley and D. Kearney, "The First Real Operating System for Reconfigurable Computers," In Proc. of the 6th Australasian Computer Systems Architecture Conference, Gold Coast, Queensland, Australia, Jan. 2001, pp.130–137.

[47] R. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," In Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, CA, Apr. 1996, pp. 126–135.

[48] Xilinx, Inc. "Virtex-II Pro FPGAs,"
http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/
virtex_ii_pro_fpgas/index.htm

[49] Xilinx Inc. "ISE foundation,"
http://www.xilinx.com/ise/logic_design_prod/foundation.htm

[50] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly Coupled Reconfigurable Functional Unit," In Proc. of the 27th Annual International Symposium on Computer Architecture, Jun. 2000, pp. 225–235.

List of Publications

Journal Papers

- M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima and K. Watanabe, “An Efficient and Effective Algorithm for Online Task Placement with I/O Communications in Partially Reconfigurable FPGAs,” IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E89-A, No. 12, pp. 3416-3426, Dec. 2006.

International Conference

- M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima and K. Watanabe, “Online Task Placement for Partially Reconfigurable FPGAs Using I/O Routing Information,” In Proc. of the 13th Workshop on Synthesis And System Integration of Mixed Information technologies, Nagoya, Japan, Apr. 2006, pp. 342–349.
- M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima and K. Watanabe, “A New Approach to Online FPGA Placement,” In Proc. of Conference on Information Science and Systems, Princeton, NJ, Mar. 2006, CD-ROM.
- M. Tomono, M. Nakanishi, S. Yamashita, and K. Watanabe, “Event-Oriented Computing with Reconfigurable Platform,” In Proc. of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC 2005), Shanghai, Chian, Jan. 2005, pp. 1248-1251.

Workshops and Technical Reports

- Mitsuru Tomono, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima, “A Placement and Routing Algorithm for a Reconfigurable 1- bit Processor Array,” NAIST Technical Report, NAIST-IS-TR2007004, March 2007.
- M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima and K. Watanabe, “Online FPGA Placement Using I/O Routing Information,” In

Proc. of IEICE Technical Report of VLSI Design Technologies, Ehime, Japan, May 2006, pp. 1–6.

- M. Tomono, M. Nakanishi, S. Yamashita, and K. Watanabe, "Online FPGA Placement under I/O Timing Constraints," In Proc. of IEICE Technical Report of VLSI Design Technologies, (DesignGaia 2005), Kyushu, Japan, Nov./Dec. 2005, pp. 7–12.
- M. Tomono, M. Nakanishi, S. Yamashita, and K. Watanabe, "Event-Oriented Computing with Reconfigurable Platform and its Application," In Proc. of IEICE Technical Report of 4th Reconfigurable Computing Systems Workshop, Nagasaki, Japan, Sep. 2004, pp.103–109.
- M. Tomono, M. Nakanishi, S. Yamashita, and K. Watanabe, "Dynamically Reconfigurable Coprocessors for Exception Detection," In Proc. of IEICE Technical Report of VLSI Design Technologies, Osaka, Japan, May 2004, pp. 13–18.
- M. Tomono, M. Nakanishi, S. Yamashita, and K. Watanabe, "Architecture for Active Software that can Rearrange Active Functions, " In Proc. of IEICE Technical Report of VLSI Design Technologies, (DesignGaia 2003), Kyushu, Japan, Nov. 2003, pp. 151–155.

Other Publications

Journal Papers

- K. Watanabe, A. Inoue, M. Tomono, K. Kurakawa, M. Nakanishi, and S. Yamashita, "Assertion Verification Design by Active Function," JSSST Journal on Computer Software, Vol. 22, No. 3, pp.76–91, Jul. 2005, (In Japanese).

Workshops

- K. Watanabe, A. Inoue, M. Toshiaki, M. Tomono, M. Nakanishi, and S. Yamashita, "Design of Active Software with Safety," In Proc. of AIST Programming Science Group Technical Report, Osaka, Japan, Feb. 2004, pp. 115–125.