Dynamic Reconfigurable Architecture Exploration based on Parameterized Reconfigurable Processor Model

Ittetsu TANIGUCHI, Keishi SAKANUSHI, Kyoko UEDA, Yoshinori TAKEUCHI, and Masaharu IMAI

Graduate School of Information Science and Technology
Osaka University

Abstract. In recent years, dynamic reconfigurable processor which can achieve reconfiguration with a few cycles is proposed. The fast reconfiguration makes run-time reconfiguration possible, and the run-time reconfiguration gives a new possibility to the dynamic reconfigurable processor, i.e. the dynamic reconfigurable processor can also execute partitioned independent subtasks with repeated reconfigurations and executions. However, to achieve an execution with the run-time reconfiguration, performance should be evaluated with various overheads: reconfiguration, memory accesses, etc. The overheads depend on reconfigurable architectures, and it is generally difficult to evaluate the overhead. As the overhead may critically affect the performance, designers should carefully explore design space for suitable architectures. In this paper, we propose a dynamic reconfigurable architecture exploration method based on Parameterized Reconfigurable Processor model (PRP-model) and task partitioning optimization algorithm for architecture exploration corresponding to proposed PRP-model. Experimental results showed that the proposed PRP-model and the task partitioning algorithm for PRP-model can fast evaluate various reconfigurable architectures, and designers can easily find suitable reconfigurable architectures by changing the PRP-model parameters.

1 Introduction

Portable information systems such as cellular phones and mobile MP3 players are widely spreading in our daily life. In general, there are various requirements for a design of portable information systems, e.g. low hardware cost and high performance. The requirements of low hardware cost lead to inexpensive and portable products, and the high performance is usually needed for media processing. Moreover, flexibility is required to adapt various coding standards using the same chips on board. To fulfill these requirements, system designers always design the products under the hard constraints to take account of design quality metrics: performance, area, and power. When designing embedded systems, it is essential to explore design solution space and choose a solution which they really need.

To meet these requirements, Instruction Set Processors (ISPs) or ASICs have usually been used for their design. However they cannot completely satisfy these requirements. While ISPs can flexibly execute various functions by changing software, they cannot achieve high performance. While ASICs can realize high performance for specific applications, it is difficult to use them for other applications. Therefore, as a new approach to meet them, dynamic reconfigurable processors that feature both the flexibility of ISPs and the high performance of ASICs are focused [1, 2].

Dynamic reconfigurable processors usually have many coarse grain processing elements (PEs), and each PE is connected each other by flexible interconnections. Since many PEs are simultaneously executed, the tasks with high parallelism are effectively executed. The function of dynamic reconfigurable processor is defined by configuration data, e.g. setting information of interconnections and function of each PE, that is, according to the configuration data prepared before hand, the dynamic reconfigurable processor can reconfigure itself into various circuits. The reconfiguration speed depends on total amount of configuration data and the reconfigurable architecture specification, and the reconfiguration timing is decided by the reconfiguration speed.

In recent years, dynamic reconfigurable processor which can achieve a reconfiguration with a few cycles is proposed. The fast reconfiguration makes run-time reconfiguration possible, and the run-time reconfiguration gives a new possibility to the dynamic reconfigurable processor, i.e. the dynamic reconfigurable processor can also execute partitioned independent subtasks with repeated reconfigurations and executions. The run-time reconfiguration is a special feature that traditional programmable device does not have, and the authors pay attention to this feature and its potential.

However, to achieve an execution with the run-time reconfiguration, performance should be evaluated with various overheads: reconfiguration, memory accesses, etc. The overheads depend on reconfigurable architectures, and it is generally difficult to evaluate the overhead. As the overhead may critically affect the performance, designers should carefully explore design space for suitable architectures. The authors claim varieties of reconfigurable architectures and difficulty of architecture evaluation confuse designers to explore vast design space for the best solution and fast evaluation method for various reconfigurable architectures is needed.

In this paper, we propose a Parameterized Reconfigurable Processor model (PRP-model) and a task partitioning optimization algorithm for architecture exploration corresponding to proposed PRP-model. The task partitioning optimization algorithm divides tasks into subtasks to minimize execution cycles. The algorithm is applicable to various reconfigurable architectures and supports the evaluation of various architectures for specific applications by changing PRP-model parameters. To realize run-time reconfiguration, designers can easily find the suitable reconfigurable architectures.

This paper is structured as follows: section 2 gives an overview of related work and highlights our contribution. Sections 3, 4, and 5 present a Parame-

terized Reconfigurable Processor model, a port expansion DFG, and proposed algorithm, respectively. Experimental results are given in section 6. In section 7, we conclude this paper.

2 Related Work

Various reconfigurable processor architectures have been proposed, which are classified by reconfiguration granularity or reconfiguration time [1]. To use the architectures effectively, many design methods, algorithms, and applications have been studied. Especially, task partitioning problem is one of essential dilemmas to use dynamic reconfigurable processors, and so far many studies have been conducted. However, we have never seen a task partitioning problem for execution model of repeated reconfigurations and executions.

[3] proposed a HW/SW task partitioning that considered task assignment for Instruction Set Processors (ISPs). [4] proposed a task partitioning method that partitions tasks into two reconfigurable processors with different granularities. It cannot be applied to different reconfigurable processor architectures. [5] proposed a task partitioning algorithm considering reconfigurable overhead which means the number of CLBs of FPGA for the communication. [6] proposed behavior partitioning method which does high level synthesis of each task of the behavior simultaneously. It can get the optimal solution because the problem is come down to NLP, but it takes a long time even if the target application is small. [7] proposed a task partitioning method under the constraints of the number of memory ports, and can get the optimal solution. [8] considers a task partitioning limiting the number of partitioned subtasks. In this paper, we do not limit the number of partitioned subtasks. [9] proposed the task partitioning method for DRL architecture. [3–8] proposed methods for FPGA platforms. FPGA needs to store intermediate data to an external memory at a reconfiguration, because it cannot hold the data during reconfiguration. In recent years, many dynamic reconfigurable architectures with registers in the array are proposed, and the dynamic reconfigurable architectures can hold the data in the array during reconfiguration. In this paper, we propose a task partitioning method for not only FPGA but also the dynamic reconfiguration architectures which can hold the data in the array during reconfiguration.

[10, 11] proposed reconfigurable architecture exploration method. [10] proposed a design space exploration method using the task partitioning method proposed in [6]. [11] proposed ADRES architecture template, which is unique architecture consisting of VLIW processor and a PE array, and architecture exploration method only applicable to the template. In this paper, to realize the execution model of repeated reconfigurations and executions, we propose architecture exploration method which offers designers fast evaluation of various reconfigurable architectures by changing architecture parameters.

3 Parameterized Reconfigurable Processor Model

To evaluate many reconfigurable architectures in a short time, the reconfigurable processor model which covers many kinds of processing elements and memory architectures is needed. In this paper, we propose a Parameterized Reconfigurable Processor model (PRP-model) and task partitioning algorithm based on the PRP-model.

3.1 Processor Structure Model

Figure 1 illustrates the proposed PRP-model. PRP-model includes PE array arranged processing elements (PEs), internal memories with different capacity, and configuration memory to store configuration data.

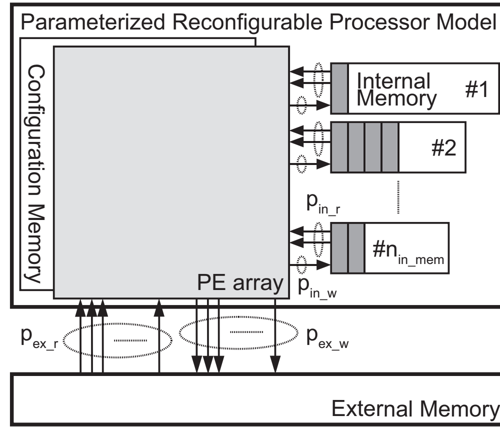

Fig. 1. Parameterized Reconfigurable Processor Model

PE Array A PE array is composed of three types of PEs: pPE, rPE, and prPE. pPE (Processing PE) has an ALU, and rPE (Register PE) has a register file, respectively. prPE (Processing and Register PE) has both an ALU and a register file. The numbers of pPEs, rPEs, and prPEs in a PRP-model are denoted as n_{pPE}, n_{rPE}, and n_{prPE}, respectively. pPEs can perform some operations, but pPEs cannot hold data at reconfiguration since it does not have any registers. rPEs can hold data at reconfiguration, but rPEs cannot perform any operations since it does not have any ALUs. Since prPEs have both ALUs and registers, it can operate and hold data. ALUs included in pPEs or prPEs have the same functionalities, and we assume that the application tasks are resolved into operations which pPEs or prPEs can execute on PEs. In this model, we treat the total amount of data as the number of data packet, and the numbers of data

that rPEs and prPEs can hold are denoted as n_{reg_rPE} and n_{reg_prPE}, respectively. In this model, we consider the available number of all PEs and not the interconnection or placement of PEs because the interconnection and placement can be defined after the number of PE is decided.

Memory Structure Memory structure of PRP-model comprises three types of memories: external memory, internal memory, and register file of rPEs or prPEs. The number of internal memory is n_{in_mem}. Each memory has some read or write access ports, and can read and write as much data as access ports at reconfiguration.

External memory is large enough to hold any data, and it has p_{ex_r} read access ports and p_{ex_w} write access ports. One read access needs t_{ex_r} cycles, and one write access needs t_{ex_w} cycles. Thus, up to p_{ex_r} data are read from external memory at t_{ex_r} cycles, and up to p_{ex_w} data are written to external memory at t_{ex_w} cycles.

The i-th internal memory keeps $n_{in_mem_dat}(i)$ data. The numbers of read port and write port of internal memory are p_{in_r} and p_{in_w}, respectively, and t_{in_r} cycles or t_{in_w} cycles are needed to read or write data.

rPE and prPE have p_{reg_r} read ports and p_{reg_w} write ports, and t_{reg_r} cycles or t_{reg_w} cycles are needed to read or write data.

In PRP-model, memory access of data read or write can be done in parallel, and execution of calculation starts after all memory accesses finish.

Configuration Memory Configuration memory can store n_{config} configuration data. All configuration data are the same size, and reconfiguration always needs t_{config} cycles. After k-th reconfiguration, a new configuration data can be overwritten to k-th configuration data from external memory. It takes t_{cfg_r} cycles to store one configuration data to configuration memory.

Let $n_{subtask}$ be the number of partitioned subtasks to execute, that is, $n_{subtask}$ configurations are needed to execute the target task. When $n_{subtask}$ is not over n_{config}, all configuration data can be kept at configuration memory. On the other hand, when $n_{subtask}$ is greater than n_{config}, $n_{subtask} - n_{config}$ configuration data cannot be kept at configuration memory. Thus, at run-time, the configuration data which cannot be kept at configuration memory are read from external memory by its reconfiguration. The configuration data read and task execution can execute simultaneously.

Usually, memory specification is defined by bit width and memory depth. Let BW_{CM} and MD_{CM} be the bit width and the memory depth of the configuration memory, respectively. The total bit of configuration memory TB_{CM} is calculated as follows:

$$TB_{CM} = BW_{CM} \cdot MD_{CM}. \quad (1)$$

Let TB_{config} be the number of total bit of one configuration data, and TB_{config} is expressed by Eq.(2).

$$TB_{config} = Scale \cdot (n_{pPE} + n_{prPE} + n_{rPE}), \quad (2)$$

where $Scale$ is a parameter of reconfigurable architecture complexity, and it is defined by each reconfigurable architecture, e.g. function of each PE, interconnection, etc. In this research, we assume the configuration data is increased linearly according to the number of PEs because the function of PE and the interconnection are not considered in PRP-model.

Then, n_{config} and t_{cfg_r} is expressed by Eq.(3) and Eq.(4).

$$n_{config} = \left\lfloor \frac{TB_{CM}}{TB_{config}} \right\rfloor . \tag{3}$$

$$t_{cfg_r} = \left\lceil \frac{TB_{config}}{BW_{CM}} \right\rceil . \tag{4}$$

3.2 Memory Access Overhead

PRP-model has storage resources: external memory, internal memory, and register file of rPEs and prPEs. The input data of application are read from external memory and the output data of application are finally written to external memory. Thus, internal memories and register files in PEs are used to keep temporal data at reconfiguration. When the temporal data cannot be kept at internal memories or register file due to the limitation of the capacity, the data are stored to external memory.

The memory access cycles on PRP-model are calculated as follows. Let $req_{in_r}(i,j)$ be the number of requested data from the i-th internal memory at configuration j. The data read cycles at configuration j, $T_{in_r}(j)$, are expressed by Eq.(5).

$$T_{in_r}(j) = \max_{0 \le i < n_{in_mem}} \left\{ T_{in_r}(i,j) \right\} , \tag{5}$$

where $T_{in_r}(i,j)$, the data read cycles from i-th internal memory at configuration j, are as follows:

$$T_{in_r}(i,j) = \left\lceil \frac{req_{in_r}(i,j)}{p_{in_r}} \right\rceil \cdot t_{in_r} . \tag{6}$$

Let $req_{ex_r}(j)$ be the number of requested data from external memory at configuration j. The data read cycles at configuration j, $T_{ex_r}(j)$, are expressed by Eq.(7).

$$T_{ex_r}(j) = \left\lceil \frac{req_{ex_r}(j)}{p_{ex_r}} \right\rceil \cdot t_{ex_r} . \tag{7}$$

Let $req_{prPE_r}(i,j)$ be the number of requested data from i-th prPE at configuration j. The data read cycles at configuration j, $T_{prPE_r}(j)$, are expressed by Eq.(8).

$$T_{prPE_r}(j) = \max_{0 \le i < n_{prPE}} \left\{ T_{prPE_r}(i,j) \right\} , \tag{8}$$

where $T_{prPE_r}(i,j)$, the data read cycles from i-th prPE at configuration j, are as follows:

$$T_{prPE_r}(i,j) = \left\lceil \frac{req_{prPE_r}(i,j)}{p_{reg_r}} \right\rceil \cdot t_{reg_r}. \qquad (9)$$

Let $req_{rPE_r}(i,j)$ be the number of requested data from i-th rPE at configuration j. The data read cycles at configuration j, $T_{rPE_r}(j)$, are expressed by Eq.(10).

$$T_{rPE_r}(j) = \max_{0 \le i < n_{rPE}} \left\{ T_{rPE_r}(i,j) \right\}, \qquad (10)$$

where $T_{rPE_r}(i,j)$, the data read cycles from i-th rPE at configuration j, are as follows:

$$T_{rPE_r}(i,j) = \left\lceil \frac{req_{rPE_r}(i,j)}{p_{reg_r}} \right\rceil \cdot t_{reg_r}. \qquad (11)$$

The data read cycles ("Memory Access (Data Read)" in figure 2) of configuration j are expressed by Eq.(12).

$$T_r(j) = \max \left\{ T_{ex_r}(j), T_{in_r}(j), T_{PE_r}(j) \right\}, \qquad (12)$$

where $T_{PE_r}(j)$, the data read cycles from prPE and rPE at configuration j, are as follows:

$$T_{PE_r}(j) = \max \left\{ T_{prPE_r}(j), T_{rPE_r}(j) \right\}. \qquad (13)$$

The data write cycles at configuration j, $T_w(j)$, are expressed the same way.

3.3 Processing and Reconfiguration

Figure 2 shows processing flow of PRP-model.

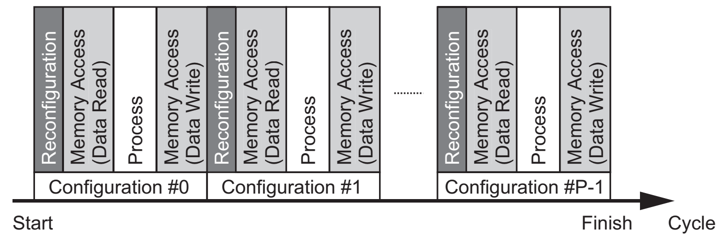

Fig. 2. Processing Flow

PRP-model is configured to the target circuit according to the configuration data, and input data or intermediate data are read from external memory,

internal memory, and register file of rPEs and prPEs. Then the partitioned subtasks are processed after reading data. Finally, output data or intermediate data are stored to storage resources. PRP-model processes this processing flow repeatedly. However, as the limited configuration memory, the processing flow of PRP-model may be stalled according to the reading cycles of the configuration data needed to the reconfiguration. The waiting cycles for configuration data read are calculated as follows.

Let $End_{exec}(i)$ and $End_{cfg_r}(i)$ be the end time of the execution of configuration i and the end time of i-th configuration data read from external memory, respectively. The end time of i-th reconfiguration is expressed by Eq.(14).

$$End_{reconf}(i) = Start_{reconf}(i) + t_{config}, \tag{14}$$

where $Start_{reconf}(i)$ is as follows:

$$Start_{reconf}(i) = \max\left\{End_{exec}(i-1), End_{cfg_r}(i)\right\}. \tag{15}$$

After i-th reconfiguration, configuration i is executed. Let $T_{proc}(i)$ be the processing cycles of i-th subtask without memory access cycles, and the end time of execution of configuration i are expressed by Eq.(16).

$$End_{exec}(i) = Start_{exec}(i) + T_r(i) + T_{proc}(i) + T_w(i), \tag{16}$$

where $Start_{exec}(i)$ is as follows:

$$Start_{exec}(i) = End_{reconf}(i). \tag{17}$$

PRP-model can simultaneously read the configuration data and execute the task. The start time of configuration data read is expressed by Eq.(18).

$$Start_{cfg_r}(i) = \max\left\{End_{cfg_r}(i-1), End_{reconf}(i - n_{config})\right\}, \tag{18}$$

where $End_{cfg_r}(i)$ is as follows:

$$End_{cfg_r}(i) = Start_{cfg_r}(i) + t_{cfg_r}. \tag{19}$$

Therefore, the waiting cycles according to the stalled processing flow are expressed by Eq.(20).

$$T_{wait} = \sum_{i=0}^{P-1} \left\{Start_{reconf}(i+1) - End_{exec}(i)\right\}. \tag{20}$$

Total execution cycles are evaluated including T_{wait}.

4 Port Expansion DFG

We define port expansion DFG to evaluate reconfiguration overhead in terms of exactly calculating the number of memory accesses. Data flow graph (DFG)

is one method of task representation with execution dependency. Traditional DFG is composed of nodes and edges. Node represents an operation, and the edge connected to a pair of nodes means the data flow between them. Incoming edges of a node mean the input data used for the corresponding operation, and outgoing edges of a node mean the results from corresponding operation.

When the data are used at some nodes, data read sometimes occurs but data write occurs only one time. The traditional DFG cannot precisely represent the number of memory accesses because the differences of the data represented by outgoing edges cannot be distinguished. Thus, we label the input/output of node of traditional DFG as “port”, which represents the input or output data. We call this DFG as port expansion DFG. Each node of the port expansion DFG has ports, and each edge connects to a pair of ports.

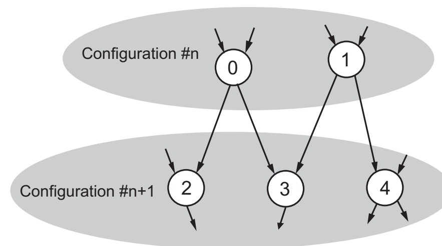

Fig. 3. Traditional DFG

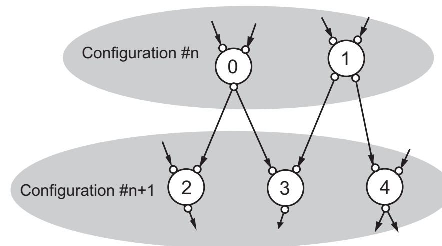

Fig. 4. Port Expansion DFG

Figure 4 shows an example of a port expansion DFG based on a traditional DFG showed in figure 3. In figure 3, it is difficult to calculate the number of memory accesses when the configuration n is reconfigured into the configuration $n + 1$. However, using a port expansion DFG, we can recognize that node 0 clearly outputs data used at nodes 2 and 3, and node 1 outputs two data that are respectively used at nodes 3 and 4. Thus, it can be obtained that three data writes are required after configuration n.

5 Task Partitioning Algorithm

5.1 Task Partitioning Problem

In this section, we define the task partitioning problem for PRP-model.

– Task Partitioning Problem –

For given a port expansion DFG of the target application and a PRP-model, to find a task partition and storage resource assignment whose execution cycles are minimum keeping the execution order defined by port expansion DFG.

□

5.2 Outline of Task Partitioning Algorithm

In this section, we propose a task partitioning algorithm using Simulated Annealing (SA) for PRP-model that consists of configuration and storage resource assignments. The following is the outline of task partitioning algorithm.

1. Initial solution decision.
2. The following steps are repeatedly processed until SA's final condition is satisfied:
   (a) Configuration assignment by MOVE operation.
   (b) Storage resource assignment.
   (c) Execution cycles estimation for the current solution.
   (d) Optimal solution update.

In configuration assignment, the task is partitioned into several subtasks, and each subtask is assigned to a configuration according to an execution order. Then in storage resource assignment, the intermediate data between configurations are assigned to storage resources.

5.3 Configuration Assignment

In this section, we explain how to make neighbor solution (configuration assignment) using MOVE operation. The MOVE operation in SA is the movement of a randomly selected node to an adjacent configuration. We define two MOVE operations: $MOVE_{BWD}$ and $MOVE_{FWD}$. $MOVE_{BWD}$ is the movement of the node to the previous configuration, and $MOVE_{FWD}$ is the movement of the node to the next configuration (figure 5).

Let $Child(x)$ and $Parent(x)$ be a set of child nodes of node x and a set of parent nodes of node x, respectively. Let $Cfg(s, x)$ and $Vacant(s, c)$ be the number of configuration assigned to node x of the solution s and the number of vacancies of configuration c of the solution s, respectively. We call configuration c *empty* when there is no nodes in the configuration c.

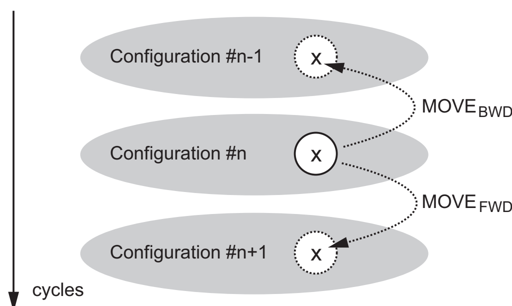

Fig. 5. $MOVE_{BWD}$ and $MOVE_{FWD}$ operations

When the node n in the solution s satisfies Eq.(21) and Eq.(22), $MOVE_{BWD}$ can apply to node n.

$$\forall x \in Parent(n); Cfg(s,x) < Cfg(s,n) \tag{21}$$

$$Vacant(s, Cfg(s,n) - 1) > 0 \tag{22}$$

Similarly, when the node n in the solution s satisfies Eq.(23) and Eq.(24), $MOVE_{FWD}$ can apply to node n.

$$\forall x \in Child(n); Cfg(s,x) > Cfg(s,n) \tag{23}$$

$$Vacant(s, Cfg(s,n) + 1) > 0 \tag{24}$$

$MOVE_{BWD}$ and $MOVE_{FWD}$ operations occur sometimes an *empty* configuration. In figure 6(a), there is *empty* configuration c in the configuration sequence because of $MOVE_{BWD}$ for the node k. In such case, $MOVE_{BWD}$ removes the *empty* configuration c after moving node k (figure 6(b)). We call this operation "Packing". By the packing, the number of the configuration is not always greater than the number of nodes of port expansion DFG.

5.4 Storage Resource Assignment

In this section, we define storage resource assignment.

"Data1" and "data2", illustrated in figure 7, should be saved as storage resources because these data cannot be kept at reconfiguration. Thus, the storage resource assignment algorithm assigns these interconfiguration data to storage resource according to the following policy after configuration assignment:

- Data are assigned to the highest priority resource with space.

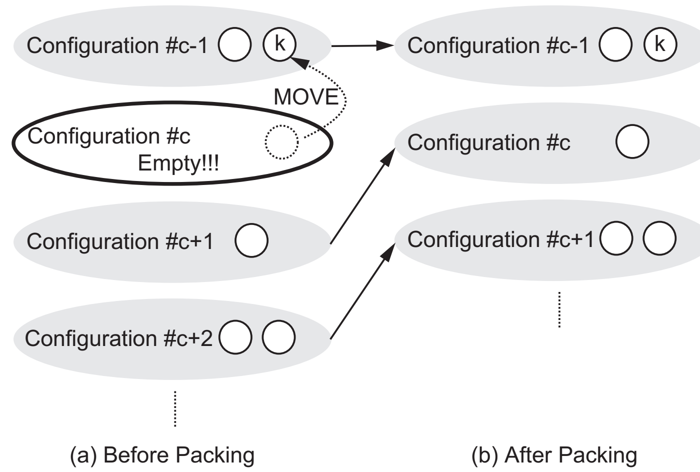

Fig. 6. Packing for *empty* configuration

– The first priority resource is rPE, the second is prPE, the third is internal memory, and last is external memory.

Figure 8 shows an example of storage resource assignment, and "data1" and "data2" are stored external memory and internal memory at reconfiguration, respectively. Note that memory accesses to read or write are needed at reconfiguration. When the assigned data become unnecessary, other data can be overwritten.

Since the number of interconfiguration data is changed when the configuration assignment is changed by the MOVE operation, the storage resource assignment is recalculated after the MOVE operation.

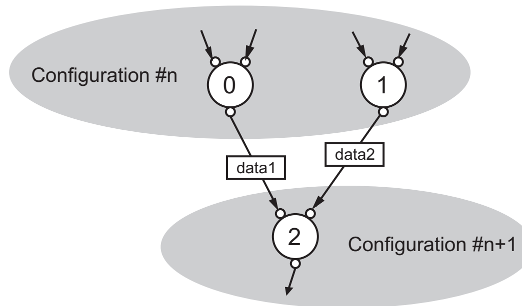

Fig. 7. The Data Crossed Border of Two Configurations

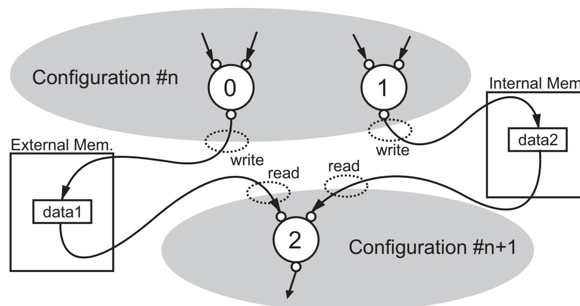

Fig. 8. Example of Storage Resource Assignment

6 Experiment

To demonstrate the efficiency of our proposed method, we show two experimental results: evaluation of task partitioning algorithm for PRP-model and reconfigurable architecture exploration. The experimental environment includes PentiumD 2.8 GHz, 2 GB memory, and Fedora 4. The Simulated Annealing (SA) parameters were as follows: initial temperature was 10, final temperature was 0.01, and the cooling ratio of the temperature was 0.98. Solutions obtained by SA are according to SA's random seed. Thus, we evaluate the average execution cycles and CPU times of 10 solutions obtained by different random seeds.

6.1 Evaluation of Task Partitioning Algorithm

To demonstrate the efficiency of the proposed algorithm, we applied it to two applications: DCT8 and CHENG. DCT8 is a one-dimensional eight point discrete cosine transform (DCT), implemented completely in parallel. CHENG is Cheng's DCT algorithm implemented completely in parallel, too. The number of nodes in DCT8's port expansion DFG is 50, which is 13 in CHENG. We evaluate the proposed task partitioning algorithm by focusing on the quality of solutions and CPU time. We evaluated 8 architectures under the variety of n_{pPE}, n_{prPE}, and n_{config}. The other architecture parameters were fixed and described in Table 1.

Table 1. Fixed Parameters

Parameter	Value	Parameter	Value
t_{ex_r}	2	t_{ex_w}	3
t_{in_r}	1	t_{in_w}	2
t_{reg_r}	1	t_{reg_w}	1
p_{ex_r}	4	p_{ex_w}	4
p_{in_r}	1	p_{in_w}	1
p_{reg_r}	1	p_{reg_w}	1
n_{in_mem}	0	n_{rPE}	0
n_{reg_prPE}	1	t_{config}	1
t_{cfg_r}	16		

Table 2 shows comparison results of the execution cycles of target application CHENG. "Proposed" in Table 2 is execution cycles obtained by the proposed task partitioning algorithm, and "Opt." is the optimal execution cycles obtained by branch and bound strategies. Experimental results showed that for all architectures the proposed task partitioning algorithm can obtain the same execution cycles as the optimal solutions. Notice that execution cycles decreased in Table 2 when the number of PEs changed from four to eight, because the more operations that are executable simultaneously, the fewer the reconfigurations. Using prPEs with internal register files decreases the execution cycles more than using pPE. Reconfiguration overhead is decreased because prPEs have register files that pPEs do not have at storage resource assignment.

Table 2. Comparison of Execution Cycles

No.	Parameters			Architecture		Proposed	Opt.
	n_{pPE}	n_{prPE}	n_{config}				
1	0	4	1	4 prPEs	with Config. Mem. (1 Config.)	57	57
2	0	4	2		with Config. Mem. (2 Config.)	39	39
3	0	8	1	8 prPEs	with Config. Mem. (1 Config.)	25	25
4	0	8	2		with Config. Mem. (2 Config.)	16	16
5	4	0	1	4 pPEs	with Config. Mem. (1 Config.)	58	58
6	4	0	2		with Config. Mem. (2 Config.)	40	40
7	8	0	1	8 pPEs	with Config. Mem. (1 Config.)	27	27
8	8	0	2		with Config. Mem. (2 Config.)	20	20

Next, we compared CPU time under the same conditions as previous and target applications, CHENG and DCT8. Table 3 shows CPU time comparisons. "Proposed" is CPU time by the proposed algorithm, and "Opt." is CPU time to get an optimal solution by branch and bound strategies. Table 3 shows that the proposed algorithm can obtain CHENG's solution in 30.7 seconds in the worst case when the optimal solution is obtained in about 57 minutes (3391 sec.). Furthermore, when DCT8 is the target application, the optimal solution cannot be obtained in practical time. However the proposed algorithm can obtain solutions in about 150 sec. in the worst case. Thus, the proposed algorithm can obtain solutions for various architectures in practical time.

Table 3. CPU Time

No.	Architecture		CHENG		DCT8	
			Proposed [sec]	Opt. [sec]	Proposed [sec]	Opt. [sec]
1	4 prPEs	with Config. Mem. (1 Config.)	30.7	3391	153.8	NA
2		with Config. Mem. (2 Config.)	30.5	3398	153.0	NA
3	8 prPEs	with Config. Mem. (1 Config.)	29.1	4595	140.1	NA
4		with Config. Mem. (2 Config.)	29.3	4597	139.9	NA
5	4 pPEs	with Config. Mem. (1 Config.)	27.1	3202	140.2	NA
6		with Config. Mem. (2 Config.)	27.1	3201	140.3	NA
7	8 pPEs	with Config. Mem. (1 Config.)	26.0	4220	120.8	NA
8		with Config. Mem. (2 Config.)	25.9	4222	120.8	NA

Table 4. CPU Time for Complex Example [sec]

	$n_{pPE} = 8$	$n_{pPE} = 64$	$n_{pPE} = 256$
$N = 100$	310.9	242.5	242.2
$N = 300$	1390.7	915.7	811.4
$N = 500$	3160.3	1690.8	1427.5

Table 4 shows CPU time for more complex port expansion DFGs under the same conditions as previous. The number of port expansion DFG's nodes N equals 100, 300, and 500, and the number of pPE n_{pPE} equals 8, 64, and 256. In table 4, the worst search time is about 50 minutes (3160.3 sec.). Thus proposed algorithm can get the solution for complex input in the practical time with various architecture parameters.

However, from solution's quality perspective, proposed algorithm cannot always obtain a good solution. In case of $N = 500$ and $n_{pPE} = 256$, the number of configurations results in four, and this solution is not feasible. Analyzing this result, some configurations can be merged, and the execution cycles can be drastically reduced by decreasing number of configurations. When the number of PEs is $n_{pPE} = 256$, the same phenomenon occurs. The reason why proposed algorithm cannot obtain the good solution is discontinuity of solution space. The discontinuity of solution space prevents proposed algorithm from efficient search for a good solution. To solve bigger problem more than 128 PEs, the proposed algorithm should be modified.

6.2 Reconfigurable Architecture Exploration

In this section, we demonstrate reconfigurable architecture exploration using PRP-model under the variety of the number of PEs and configuration memories. Target application is the sample DFG, whose number of nodes equals to 500, used previous experiment. Experimental environment is the same conditions as previous. The above-mentioned experimental results show that proposed algorithm cannot always obtain a good solution in case of more than 128 PEs. Therefore, we demonstrate reconfigurable architecture exploration under the number of PEs from 16 to 128.

Table 5 shows fixed parameters of explored reconfigurable architectures, and table 6 shows specifications of configuration memories. We assume the parameter of reconfigurable architecture complexity $Scale$ in Eq.(2) equals 128. Then, n_{config} and t_{cfg_r} are shown in Table 7. Table 7 shows that n_{config} is decreased half when the number of PEs is increased two times.

Figure 9 shows the execution cycles of each architecture with configuration memory A, B, and C, and the number of $prPE$ equals zero. In Figure 9, the execution cycles simply increase/decrease in the case of configuration memory A/C according to the increase of the number of PEs. On the other hand, in the case of configuration memory B, the execution cycles progressively decrease according to the increase of the number of PEs. However, when the number of

Table 5. Fixed Parameters for Reconfigurable Architecture Exploration

Parameter	Value	Parameter	Value
t_{ex_r}	2	t_{ex_w}	3
t_{in_r}	1	t_{in_w}	2
t_{reg_r}	1	t_{reg_w}	1
p_{ex_r}	4	p_{ex_w}	4
p_{in_r}	1	p_{in_w}	1
p_{reg_r}	1	p_{reg_w}	1
n_{in_mem}	0	n_{rPE}	0
n_{reg_prPE}	1	t_{config}	1

Table 6. Configuration Memory Specification

Config. Mem. Type	Total Bit TB_{CM} [bit]	Memory Depth MD_{CM}	Bit Width BW_{CM} [bit]
A	32k	2k	16
B	64k	4k	16
C	128k	8k	16
D	32k	1k	32
E	64k	2k	32
F	128k	4k	32

Table 7. n_{config} and t_{cfg_r} ($Scale = 128$)

#PEs	Configuration Memory Type A		Type B		Type C		Type D		Type E		Type F	
	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}
16	16	128	32	128	64	128	16	64	32	64	64	64
32	8	256	16	256	32	256	8	128	16	128	32	128
64	4	512	8	512	16	512	4	256	8	256	16	256
128	2	1024	4	1024	8	1024	2	512	4	512	8	512

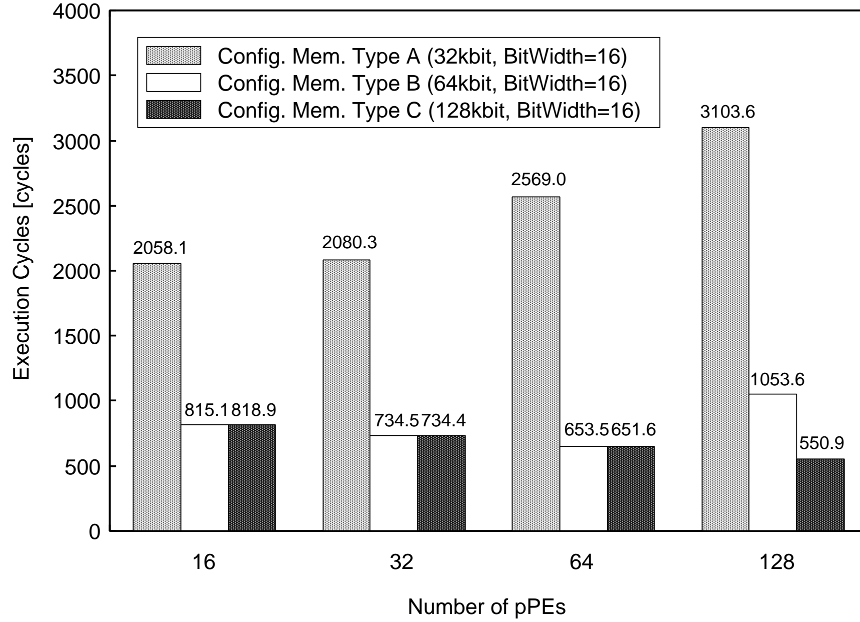

Fig. 9. Execution Cycles ($Scale = 128$, Config. Mem. Type is A, B, and C.)

PEs equals 128, the execution cycles increase greater than the execution cycles when the number of PEs equals 16.

Figure 10 shows the execution cycles of each architecture with configuration memory D, E, and F. We can also see that the execution cycles simply increase/decrease in the case of configuration memory D/F according to the increase of the number of PEs. The increasing rate of execution cycles is more slowly than Figure 9. At the same time, the execution cycles always decrease according to the increase of the number of PEs in the case of configuration memory E, whose total bit equals configuration memory B's one.

When the total bit of configuration memory is the same, the bit width of configuration memory affects the increase of the execution cycles. The increase of configuration memory bit width leads to the decrease of configuration data read cycles, i.e. the overhead of configuration data read decreases. For less overhead, designers should choose the configuration memory which is as wide bit width as possible.

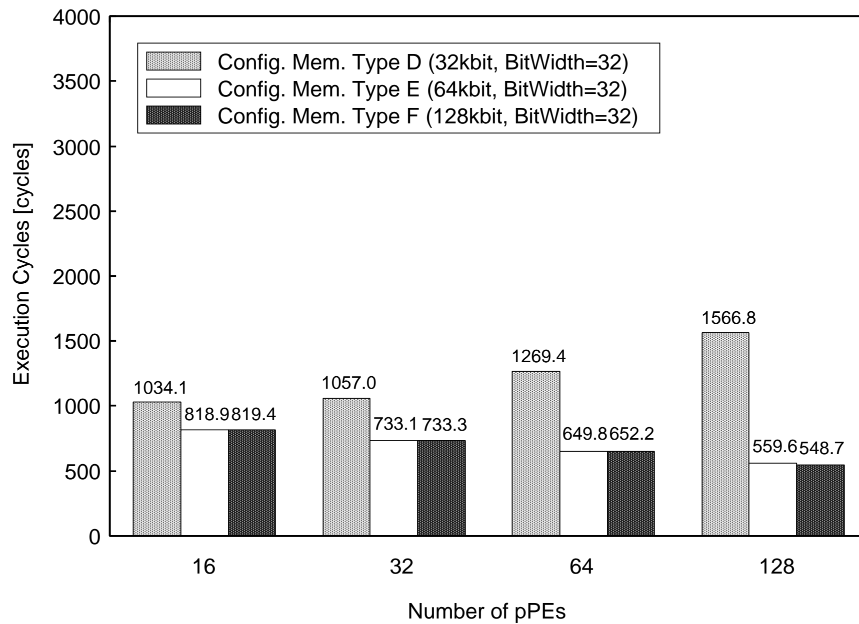

Fig. 10. Execution Cycles ($Scale = 128$, Config. Mem. Type is D, E, and F.)

Table 8 shows the execution cycles of each architecture and the ratio of waiting cycles included the execution cycles. In the increasing case of the execution cycles, we can see that waiting cycles account for a large percentage of the execution cycles. On the other hand, in the decreasing case, the ratio of waiting cycles always equals zero.

Table 8. Total Execution Cycles and Waiting Ratio (pPE Case)

#PEs	Configuration Memory											
	Type A		Type B		Type C		Type D		Type E		Type F	
	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]
16	2058.1	58.5	815.1	0.0	818.9	0.0	1034.1	17.6	818.9	0.0	819.4	0.0
32	2080.3	63.1	734.5	0.0	734.4	0.0	1057.0	27.0	733.1	0.0	733.3	0.0
64	2569.0	73.2	653.5	0.0	651.6	0.0	1269.4	46.2	649.8	0.0	652.2	0.0
128	3103.6	81.2	1053.6	44.2	550.9	0.0	1566.8	62.7	559.6	0.3	548.7	0.0

The above-mentioned experimental results are obtained when $prPE$ is not used. Table 9 shows the execution cycles and the ratio of waiting cycles when $prPE$ is only used. Experimental results in Table 9 show the execution cycles are 150-300 cycles less than Table 8 in the case of the ratio of waiting cycles equals zero. However, the execution cycles are almost the same as Table 8 when the ratio of waiting cycles is not zero.

Table 9. Total Execution Cycles and Waiting Ratio ($prPE$ Case)

#PEs	Configuration Memory											
	Type A		Type B		Type C		Type D		Type E		Type F	
	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]	Total	Wait[%]
16	2056.2	62.6	675.1	0.0	668.7	0.0	1032.1	25.1	677.0	0.0	668.5	0.0
32	2073.0	69.6	553.4	0.0	552.6	0.0	1048.9	40.3	560.9	0.0	545.7	0.0
64	2567.1	81.3	519.1	6.2	426.9	0.0	1287.1	63.2	427.4	0.0	424.1	0.0
128	3091.0	90.7	1043.0	72.8	234.4	0.0	1554.5	81.5	530.2	45.7	233.6	0.0

Because of the limitation of configuration memory, n_{config} simply decrease according to the increase of the number of PEs. In contrast, t_{cfg_r} simply increase according to the increase of the number of PEs because we assume the configuration data linearly increase according to the number of PEs. Therefore, the overhead of configuration data read drastically increases according to the increase of the number of PEs, and critically affects the execution cycles.

When the reconfigurable architecture which includes small amount of configuration memory is used, experimental results show that the overhead of configuration data read is dominant, and the overhead critically affects the execution cycles. To use the reconfigurable processor effectively, designers should carefully design the configuration memory and its parameters, i.e. t_{config}, n_{config}, and t_{cfg_r}.

Table 10. n_{config} and t_{cfg_r} ($Scale = 64$)

#PEs	Configuration Memory											
	Type A		Type B		Type C		Type D		Type E		Type F	
	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}	n_{config}	t_{cfg_r}
16	32	64	64	64	128	64	32	32	64	32	128	32
32	16	128	32	128	64	128	16	64	32	64	64	64
64	8	256	16	256	32	256	8	128	16	128	32	128
128	4	512	8	512	16	512	4	256	8	256	16	256

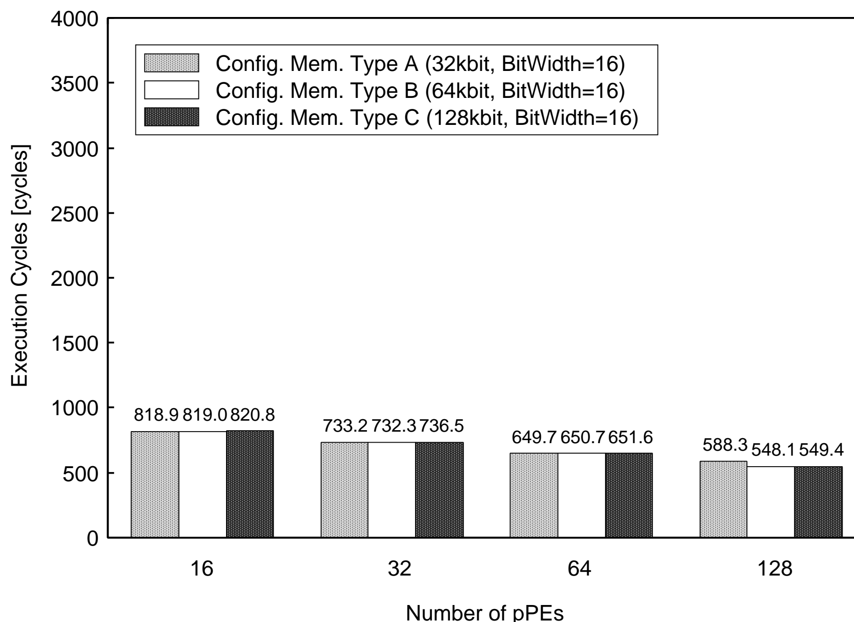

Fig. 11. Execution Cycles ($Scale = 64$, Config. Mem. Type is A, B, and C.)

Next, we assume the parameter of reconfigurable architecture complexity $Scale$ in Eq.(2) is decreased to 64, i.e. designers use simpler reconfigurable architecture. Table 10 shows n_{config} and t_{cfg_r} in case of $Scale = 64$. Figure 11 shows the execution cycles of each architecture, whose $Scale$ equals to 64, with configuration memory A, B, and C. In Figure 11, compared to Figure 9, the execution cycles always decrease according to the increase of the number of PEs because the decrease of $Scale$ makes the configuration data half. To decrease overhead, the reduction of reconfigurable architecture complexity $Scale$ also effectively affects.

When the performance is not enough, designers often add more PEs to execute effectively with rich HW. However, to use the execution with run-time reconfiguration, addition of PEs does not always improve the performance because of effect of the complex overhead. Considering the effect of the overhead carefully, the best reconfigurable architecture which satisfies the requirement should be chosen. It is the first step for the effective execution with run-time reconfiguration to analyze the details of reconfigurable architecture carefully.

7 Conclusion

In this paper, we proposed dynamic reconfigurable architecture exploration method based on Parameterized Reconfigurable Processor model and task partitioning

optimization algorithm for reconfigurable architecture exploration corresponding to proposed PRP-model. Using proposed method, designers can fast evaluate various reconfigurable architectures, and easily find suitable reconfigurable architectures by changing PRP-model parameters. Future work includes the modification of task partitioning optimization algorithm corresponding to large size architectures, the establishment of reconfigurable architecture exploration that considers area and power, and the processing element function decision method according to the total configuration data constraint.

References

1. Reiner Hartenstein. A Decade of Reconfigurable Computing: a Visionary Restrospective. In *Proc. of DATE'01*, pages 642–649, 2001.
2. Andreu Mas-Corell, Michael Winston, and Jerry Green. Introduction to Reconfigurable Hardware. In *System Level Design of Reconfigurable System-on-Chips*, pages 15–26. Springer, 2005.
3. S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration. In *Proc. of DAC'05*, pages 335–340, 2005.
4. M. D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris, and C. E. Goutis. A Methodology for Partitioning DSP Applications in Hybrid Reconfigurable Systems. In *Proc. of ISCAS'05*, pages 1206–1209, 2005.
5. Karthikeya M. Gajjala Purna and Dinesh Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Trans. on Computers*, 48(6):579–590, June 1999.
6. M. Kaul and R. Vemuri. Optimal Temporal Partitioning and Synthesis for Reconfigurable Architecture. In *Proc. of DATE'98*, pages 389–396, 1998.
7. B. Ouni, A. Mtibaa, and M. Abid. Synthesis and Time Partitioning for Reconfigurable Systems. In *Design Automation for Embedded Systems*, volume 9, pages 177–191. Springer Netherlands, 2004.
8. V. Srinivasan, S. Govindarajan, and R. Vemuri. Fine-Grained and Coarse-Grained Behavioral Partitioning with Effective Utilization of Memory and Design Space Exploration for Multi-FPGA Architectures. *IEEE Trans. on VLSI Systems*, 9(1):140–158, 2001.
9. M. Meribout and M. Motomura. A Combined Approach to High-Level Synthesis for Dynamically Reconfigurable Systems. *IEEE Trans. on Computers*, 53(12):1508–1522, 2004.
10. M. Kaul and R. Vemuri. Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs. In *Proc. of DATE'99*, pages 202–209, 1999.
11. B. Mei, A. Lambrechts, D. Verkest, Jean-Yves Mignolet, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design & Test of Computers*, pages 90–101, March-April 2005.