INSTRUCTION BUFFER MODE FOR MULTI-CONTEXT DYNAMICALLY RECONFIGURABLE PROCESSORS

Toru Sano, Masaru Kato, Satoshi Tsutsumi, Yohei Hasegawa and *Hideharu Amano*

Department of Information & Computer Science, Keio University
3-14-1 Hiyoshi, Kohokuk-ku, Yokohama, 223-8522 Japan
email: sano@am.ics.keio.ac.jp

ABSTRACT

In multi-context Dynamically Reconfigurable Processor Array (DRPA), the required number of contexts is often increased by those with low resource usage. In order to execute such contexts without wasting a context memory, we propose a new execution mode called *instruction buffer mode* in addition to the normal multi-context mode. In this mode, a configuration code from the central configuration memory is stored in the instruction buffer and executed directly. Furthermore, by exploiting a multicast method, a single configuration code loaded to the buffer can be executed by multiple processing elements in a SIMD fashion. We also investigate a mode selection policy based on simple formulas. From the result of implementation and evaluation by using a prototype DRPA called MuCCRA-1, it appears that the total execution time is reduced 12% by using the instruction buffer mode, while 12% of the semiconductor area is increased.

1. INTRODUCTION

Dynamically Reconfigurable Processor Arrays (DRPAs) [1, 2, 3, 4] have been started to be utilized as an off-load engine for various types of System-on-Chips (SoCs) in digital appliances.

In order to achieve better area- and power-efficiency compared with traditional field-programmable devices such as FPGAs, they incorporate the following properties; (1) a simple coarse grained processor consisting of an ALU, a data manipulator, a register file and other functional modules is used as a primitive processing element (PE) of an array, and (2) dynamic reconfiguration of the PE array which enables time-multiplexed execution is introduced. Some of them provide multiple sets of configuration data called hardware contexts, from now on referred as contexts, and switch them in one or a few clock cycles. A system with such a mechanism is called a multi-context DRPA.

In order to switch context quickly, the multi-context DRPA provides context memory modules in its PEs and switching elements (SEs) to hold a certain set of configuration data corresponding to each context. Since the number of contexts which can be stored in the context memory is not so large (from 4 to 64 in general) because of the semiconductor area overhead required for the context memory, the performance is severely degraded by loading new configuration data if the number of required contexts is beyond the context memory size. Virtual hardware mechanisms [5, 6] enable to transfer new configuration data to an unused part of the context memory during execution and hide the latency of the configuration data transfer. However, even if one of these mechanisms is adopted, a large task which requires more than the context memory size is difficult to execute.

Therefore, the current multi-context DRPAs are only used to execute a limited core part of an application, and it is one of the reasons why DRPAs have not overcome their competitors: SIMD (Single Instruction stream, Multiple Data streams) style accelerators and VLIW (Very Long Instruction Word) style DSPs in some application fields.

In general, PEs in an array are not always utilized in every context. For example, the degree of parallelism in initialization and summarizing the results is not usually large. This means that contexts with low PE usage exist even in applications with a large degree of parallelism in total. For such contexts, the most part of the context memory is occupied with the idle configuration. If such contexts with a small number of PEs can work in the different mode which does not use the context memory, the efficiency of context memory can be much increased.

Here, we propose a multi-context mechanism with mixed execution mode. In addition to the normal multi-context execution mode, it provides a new mode called the *instruction buffer mode* in which a configuration code from a central configuration memory is stored in the instruction buffer and executed directly. By using this mode, the context with low PE utilization ratio can be executed without wasting the context memory. Also, by combining with configuration code multicasting method [7], a single configuration code loaded to the buffer can be executed like SIMD machines.

The rest of paper is organized as follows. Section 2 introduces a concept of the instruction buffer mode. Section 3 describes the MuCCRA-1 architecture and implementation of the mixed execution mode. Finally, we evaluate the effect of this mode in section4.

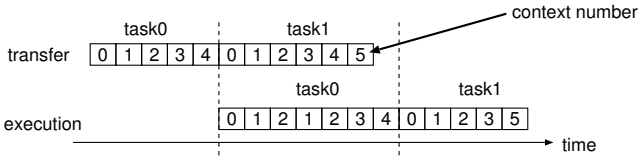

Fig. 1. Timeline of Multi-Context Mode

2. INSTRUCTION BUFFER MODE

2.1. Multi-context DRPAs

Various types of multi-context DRPAs have been developed [1, 2, 3, 8, 9], and most of them provide distributed context memory modules which hold several sets of configuration data for each PE and SE. The configuration data is read out according to a context pointer from a central control unit, and a hardware context consisting of operations of functional units in PE and data transfers using SE is switched in a clock cycle.

Since the context memory module is provided for each PE and SE in a PE array, the depth of memory module corresponding to the number of contexts is limited. For instance, three in DAPDNA-2 [3], four in FE-GA [9, 10], eight in CS2112 [8], 16 in DRP-1 [2], 32 in ADRES [1], and 64 in MuCCRA-1. The configuration data sets in the context memory are transferred from the central configuration memory before execution like Fig. 1. If the number of available context slots is not enough, the execution is suspended, and new configuration data are loaded, and then the execution is resumed. However, since it often takes hundreds of clock cycles to load a task, this step severely degrades total performance.

The virtual hardware mechanism was proposed to mitigate this problem [5, 6]. During execution of a task, the configuration data for the next task is transferred to an empty space of the context memory, and it starts immediately when the current task is finished. However, it only works efficiently when the execution time for the current task is long enough, and there is a sufficient space in the context memory to load the configuration data for the next task. A large task which cannot be stored in the context memory is difficult to be executed [6].

2.2. Instruction buffer mode

Evaluation results of the current multi-context DRPAs show that the number of utilized PEs varies depending on the contexts. For example, the usage of PEs of NEC's DRP-1 was less than 50% in average even in applications with a large degree of parallelism such as 2-Dimensional Discrete Cosine Transform. Note that several sequential steps can be mapped into a single context to increase the resource usage in DRP-1, but the results were not much improved. Thus, most applications include sequential steps which do not require a lot of PEs.

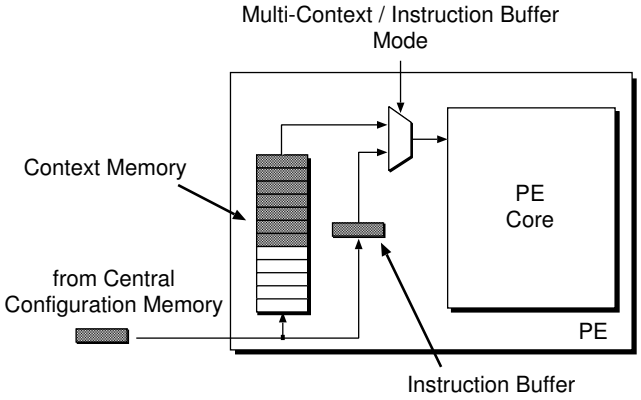

Fig. 2. Instruction buffer mode

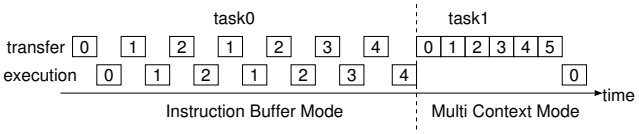

Fig. 3. Timeline of Instruction Buffer Mode

The instruction buffer mode is a mechanism to save the context memory space by using the buffer which stores and directly executes configuration code from the central configuration memory. As shown in Fig. 2, in this mode, the configuration data from the central configuration memory is not written into the context memory but stored into an additional register called *instruction buffer*, and executed immediately. Of course, it requires hundreds of clock cycles to fill the instruction buffer corresponding to the whole PE array, but if only a few PEs are used, execution can start only with a few cycles delay as shown in Fig. 3. After steps with a low PE usage are executed with the instruction buffer mode, a programmer can switch the mode, and parallel processing with a large number of PEs is done in the multi-context mode.

2.3. Configuration data multicasting

Most dynamically reconfigurable processors adopt a sequential configuration scheme giving a serial address to context memory modules and transferring configuration data to the address in order. In this method, configuration speed is limited by configuration data bus width, and the configuration data for one or a few PEs can be transferred in a clock cycle.

However, if multiple PEs use the same configuration data, the time for data transfer can be reduced by multicasting. RoMultiC [7] uses row and column multicast bits to specify configured targets instead of mapping a sequential address to PEs/SEs. As shown in Fig.4, configuration data is received by the targets where the row and column multicast bits are both '1'.

The configurable area is restricted in a rectangle, but by devising the transfer order, any complex configuration pat-

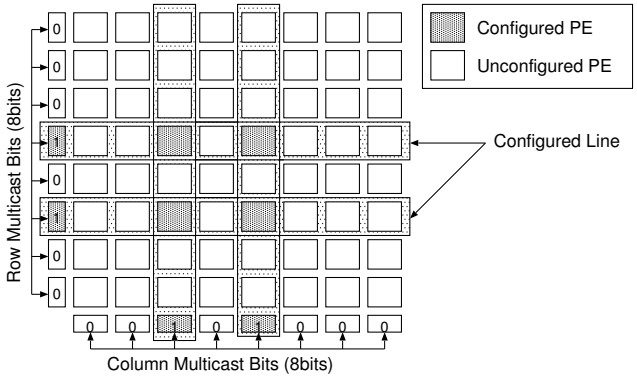

Fig. 4. RoMultiC

tern can be configured because the configuration data can be overwritten and the latest configuration data are valid. In addition, with this feature, RoMultiC can reduce further configuration data transfer cycles by scheduling their order and configuration patterns.

By using RoMultiC with the instruction buffer mode, the SIMD-like instruction can be used. That is, if PEs specified as a rectangle shape with RoMultiC multicast bits execute the same operation, they can work just after a clock cycle to transfer the same configuration data to the instruction buffer. With this function, the number of contexts can be reduced without degrading the parallelism.

2.4. Mode selection policy

Since the instruction buffer mode increases the execution time, the mode selection must be done with enough consideration. If a task contains just a single inter context loop, the total execution cycle including transfer cycles for a task in the case of both modes; $Cycle_{multi}$ and $Cycle_{instbuf}$ can be roughly estimated.

Here, let $Conf_{task}$ be a total number of configuration data of the task, $Context_{loop}$ be the number of contexts in the inter context loop, and $Context_{seq}$ be that for the rest of part, $N_{iteration}$ be the number of iterations, respectively.

$Cycle_{multi}$ can be expressed in the following equation:

$$Cycle_{multi} = Cycle_{transfer} + Cycle_{execution}, \quad (1)$$

where

$$Cycle_{transfer} = \begin{cases} 0 & \text{if the configuration data} \\ & \text{remain in the context memory} \\ Conf_{task} & otherwise \end{cases} \quad (2)$$

and

$$Cycle_{execution} = Context_{seq} + (N_{iteration} \cdot Context_{loop}). \quad (3)$$

In order to decide $Cycle_{instbuf}$ easily, we define $Conf_{context}$ as the average number of configuration data per a context with the following equation:

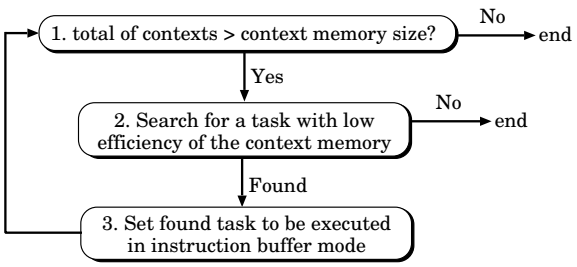

Fig. 5. Search process flow for instruction buffer mode task

$$Conf_{context} = \frac{Conf_{task}}{Context_{task}}. \quad (4)$$

$Context_{task}$ is given by $Context_{seq} + Context_{loop}$. Thus,

$$Cycle_{instbuf} = Conf_{context} \cdot Cycle_{execution}. \quad (5)$$

In order to reduce the total execution cycles, configuration data should be kept in the context memory within the time computed with expression (2), and the instruction buffer mode helps it. However, equation (5) shows that the execution cycles of the instruction buffer mode is tightly depending on the number of iterations, and so a task with a large iteration will much increase the execution cycles. Thus, a programmer must select carefully which tasks are executed in the instruction buffer mode with the following steps illustrated in Fig.5.

1. If the total sum of all tasks' contexts is less than the size of the context memory, obviously there is no reason to execute in the instruction buffer mode.
2. Search for a task which does not use the context memory efficiently compared to other tasks. Usage of the context memory, $Cycle_{context}$, is given by the following equation(6):

$$Cycle_{context} = \frac{Cycle_{execution}}{Context_{task}}. \quad (6)$$

   $Cycle_{context}$ means that how many times a configuration data is loaded from the context memory averagely.
3. If a low usage task is found, the task should be executed with the instruction buffer mode. Iterate the above steps until the number of contexts is less than the context memory size.

3. IMPLEMENTATION EXAMPLE

We added the instruction buffer mode to MuCCRA-1 [11], a prototype multi-context DRPAs. Although the MuCCRA-1 chip is now under re-designed to fix bugs in the layout, the successor chip MuCCRA-2 [11] is now available.

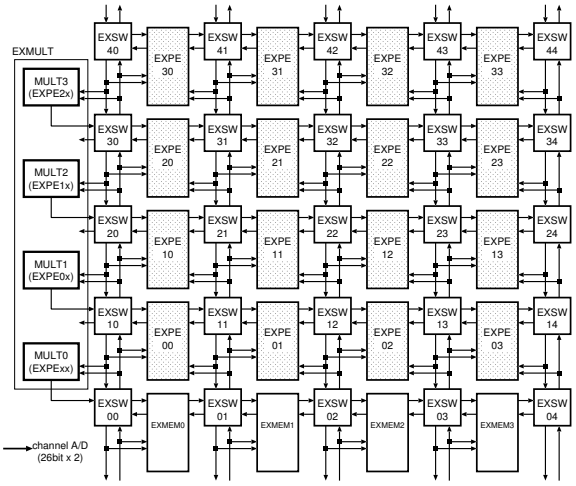

Fig. 6. PE Array Architecture of MuCCRA-1

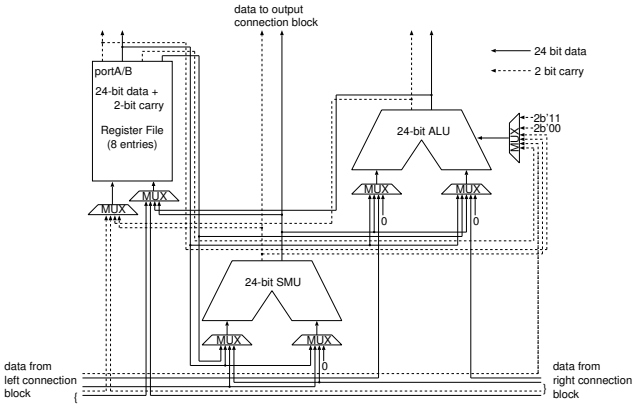

Fig. 7. PE Core Architecture of MuCCRA-1

3.1. Overview of MuCCRA-1 Architecture

Fig.6 shows the PE array structure of MuCCRA-1. The granularity of the whole architecture is 24 bits so as to fit multi-media processing, that is, all functional units and channels treat 24-bit data except wires for 2-bit carry. MuCCRA-1 has a 4×4 PE array, four multipliers (MULT) on the left side of the array and four distributed memories (MEM) which has 24-bit $\times$ 256 entries at the bottom of the array. An island-style interconnection structure like traditional FPGAs is adopted; each PE is surrounded by programmable routing wire segments. Connection blocks are provided between PEs and global routing channels for sending or receiving to or from PEs. At the intersection of vertical and horizontal channels, a Switching Element (SE) is placed. The SE is a set of simple programmable switches in which an entering channel is connected to the other SEs. There are two channels for the global routing resources.

Each PE has a programmable PE Core, connection blocks, and a context memory. In the PE Core as shown in Fig.7, like a lot of existing DRPA devices, a data manipulator called Shift & Mask Unit (SMU), an Arithmetic Logic Unit (ALU), and a register file (RFile) are provided. RFile has 26-bit (24 data bits + 2 carry bits) $\times$ 8 entries. A context memory which is 64-bit $\times$ 64-entry holds the configuration data which is distributed from the configuration memory at the beginning of an execution.

Each PE is connected with global routing wires via connection blocks. The connection blocks pick up the data from global routing wires and distribute to all the functional units of a PE Core. The operation of each functional unit and local intra-PE connection are statically defined by configuration data called a context.

Each SE consists of two multiplexer-based programmable switches (SWs) and a context memory containing configuration data which specifies a destination of each SW.

3.1.1. Context Switching Mechanism

Like the other multi-context DRPAs, each PE and SE in MuCCRA-1 equip their context memory in which the configuration data for a particular operation is held. The depth of the context memory is 64, and thus, 64 hardware contexts can be held. The central controller broadcasts a context pointer to all of the reconfigurable elements including PEs and SEs. The configuration data for a context is read out from the context memory according to the context pointer, and they are reconfigured in parallel.

3.1.2. Configuration Data Distribution Mechanism

As described in the previous section, for high speed configuration data distribution, a multicast mechanism called RoMulTiC[7] is adopted. The context control and configuration data distribution mechanism are common in all MuCCRA chips and cannot be changed except the size of context memory which influences the area of PE and SE.

3.2. Implementing the instruction buffer mode

We implemented the instruction buffer with a collection of flip-flops since just an entry is required. A signal called *mode flag* decides the target into which the configuration data is written: the context memory or the instruction buffer. The mode flag is generated by the controller consisting of the state machine which works as shown in Fig.8. First, the controller reads information containing the mode flag of a task which is about to be executed. When the mode flag is '0', it means that the task will be executed in the multi-context mode, and configuration data for all contexts of the task must be transferred into the context memories before execution. During the execution, the controller pre-loads the configuration data of the task to be executed. Otherwise, when the mode flag is '1', just one configuration data is transferred to the instruction buffer and executed immediately. The controller repeats this process until the task finished.

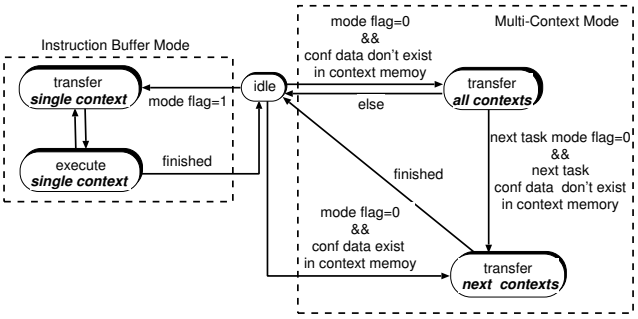

Fig. 8. State transition of the MuCCRA-1 controller

Table 1. Area Comparison

	total area (um^2)	flip-flops area ratio
original MuCCRA-1	2,761,169	15.6%
with proposed mode	3,089,248	18.9%

3.3. Hardware overhead

The MuCCRA-1 with the instruction buffer mode is also implemented using Verilog-HDL, and verified with Cadence NC-Verilog 5.83. In order to evaluate area overhead compared to MuCCRA-1 with the instruction buffer mode, we synthesized the RTL code with Rohm CMOS 180nm process technology by Synopsys Design Compiler 2007.03-SP4.

As shown in Table1, the total area of MuCCRA-1 with the instruction buffer mode increases about 11.9% compared to the original one. The main hardware increase comes from flip-flops for the instruction buffer and multiplexers implemented in all modules, PEs, SEs, MULTs, MEMs and STC.

In order to reduce this overhead, it is possible to use an entry of the context memory as the instruction buffer. In this method, a transferred instruction overwrites another task's context. However, a PE can begin to process the task after only the overwritten data is transferred again.

4. PERFORMANCE EVALUATION

4.1. Execution cycles in the case of a single task

We evaluated the execution cycles in the instruction buffer mode using three applications: 2D-DCT(Discrete Cosine Transform), Alpha-Blender, SHA-1, shown in Table 2. The term "block size" in the table means the target data size used in the calculation, and other parameters have been explained in section 2. Generally, 2D-DCT application could be implemented as 2 tasks, 1D-DCT and Transposition, but our current implementation, row and column direction of 1D-DCT are made as independent tasks, since some parameters are different from each other. And SHA-1 contains 2 inter context loop. We manually implemented these applications with a mapping tool which we developed.

In order to analyze effect of the instruction buffer mode, we evaluated the case when only a task is executed. From the implementation, $Cycle_{instbuf}$ can be estimated by expression (5), and $Cycle_{context}$ is from equation (6). They are also shown in Table 3. Since the ratio, $Cycle_{instbuf}/Cycle_{multi}$ is roughly proportional to $Cycle_{context}$, it appears that the estimated $Cycle_{instbuf}$ can be used to decide which tasks are processed in the instruction buffer mode. The errors between the results and the estimated values of Alpha-Blender and SHA-1 are relatively large. It is caused by complicated branch structure and considerable variation of PE usage between contexts in these tasks.

Table 3. Performance Results of Single Task

Task	$Cycle_{multi}$	$Cycle_{instbuf}$ (estimation)	$Cycle_{context}$
1D-DCT(row)	237	1106 (1008)	6.9
Transposition	227	208 (208)	1.0
1D-DCT(column)	244	1114 (1056)	6.5
Alpha-Blender	711	6406 (5120)	80.3
SHA-1	670	3473 (5368)	21.5

4.2. Execution cycles in the case of multiple tasks

Table 4 shows execution cycles of the following three cases when five tasks listed in Table 3 are executed sequentially 10 times one by one.

- Case 0: all five tasks are executed in the multi-context mode. Total number of contexts is 72. It exceeds the context memory size of 64 entries.
- Case 1: the transposition task is executed in the instruction buffer mode. In this case, the total number of contexts decreases to 55.
- Case 2: 8 contexts for initialization of SHA-1 is executed in the instruction buffer mode.

From the results of Case 0, it appears that the background transfer mechanism in the MuCCRA-1 works efficiently because it takes more than 8000 cycles just for transferring configuration data. Furthermore, in the both Case 1 and Case 2, although cycles for computation are increased, we achieved 3-11.7% improvement of the total execution cycles including the configuration data transfer, since all contexts could be held in the context memory. Here, any data path optimization for the instruction buffer mode were not applied in every case. Therefore, optimization could achieve further performance improvement.

5. RELATED WORK

There are several researches based on the same motivation as the instruction buffer mode.

IMEC ADRES can use the first row of PE array as a VLIW processor [1] to execute tasks with small parallelism

Table 2. Application Properties

Task	block size[bits]	$Conf_{task}$	$Context_{task}$	$Context_{sequential}$	$Context_{loop}$	iteration time	$Conf_{context}$
1D-DCT(row)	1536	146	13	2	11	8	11.2
Transposition	1536	208	17	17	0	0	12.2
1D-DCT(column)	1536	151	14	2	12	8	10.8
Alpha-Blender	8192	67	8	3	5	128	8.4
SHA-1	512	244	20	12	2/6	20/75	12.2

Table 4. Performance Results of Sequential Tasks

Case	total cycle	transfer cycle	execution cycle
0	15,316	2,666	12,650
1	14,857	297	14,560
2	13,527	607	12,920

without wasting the contexts. Compared with this approach, the instruction buffer mode can select all PEs as the target which support SIMD operations. In Morphosys [12], a certain rows or columns of PEs work in the SIMD mode. However, since the same configuration data is used for all PEs in a row or a column, the flexibility is limited. The instruction buffer mode based on RoMultiC can select PEs which work with the same instruction more flexible than the simple row and column selection in Morphosys.

By introducing the instruction buffer mode, the multi-context style DRPA can combine the ability of configuration data delivery style DPRAs like PACT XPP [4]. As the implementation example shows, it can introduce advantages of configuration data delivery style as well as SIMD operations with a small hardware overhead.

6. CONCLUSION

In this paper, the instruction buffer mode is proposed for efficient use of context memory in multi-context DRPAs. In this mode, a configuration code from the central configuration memory is stored in the instruction buffer and executed directly. Adding this mode to the normal multi-context mode results in the improvement of the context memory usage.

The evaluation results revealed that the total execution cycles are reduced by 3-12% without any modification of the applications, while 12% of the semiconductor area is increased. We also investigated a mode selection policy based on simple formulas.

As a future work, we will implement this mode on the next version of MuCCRA chip and evaluate the impact to power consumption. The future work includes the design tool which supports the SIMD execution for the instruction buffer mode for more performance improvement.

Acknowledgments: This work is supported in part by Japan Science and Technology Agency (JST). The authors thank to VLSI Design and Education Center (VDEC).

7. REFERENCES

[1] F. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom Implementation of the Coarse-Grained Reconfigurable ADRES Architecture for Multimedia Purposes," in *Proc. of FPL*, Aug. 2005, pp. 106–111.

[2] M. Motomura, "A Dynamically Reconfigurable Processor Architecture," *Microprocessor Forum*, Oct. 2002.

[3] T. Sugawara, K. Ide, and T. Sato, "Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology," *IEICE Trans. on Information & System*, vol. E87-D, no. 8, pp. 1997–2003, May 2004.

[4] M. Petrov, et al., "The XPP Architecture and Its Co-simulation within the Simulink Environment," in *Proc. of FPL*, Aug. 2004, pp. 761–770.

[5] X.-P. Ling, H. Amano, "WASMII: A Data Driven Computer on a Virtual Hardware," in Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines(FCCM'93). Springer, Berlin, 1993, pp. 33–42.

[6] H. Amano et al., "Techniques for Virtual Hardware on a Dynamic Reconfigurable Processor -An approach to tough cases-," in *Proc. of FPL*, Sept. 2004, pp. 464–473.

[7] V. Tanbunheng, M. Suzuki, and H. Amano, "RoMultiC: Fast and Simple Configuration Data Multicasting Scheme for Coarse Grain Rec onfigurable Devices," in *Proc. of FPT*, Dec. 2005, pp. 129–136.

[8] X.Tang, M.Aalsma, and R.Jou, "A compiler directed approach to hiding configuration latency in Chameleon Processors," in *Proc. of FPL*, Sept. 2000, pp. 29–38.

[9] T.Kodama, et al., "Flexible Engine: A Dynamic Reconfigurable Accelerator with High Performance and Low Power Consumption," in *Proc. of Int'l Symp. on Low-Power and High-Speed Chips (COOL Chips)*, Apr. 2006, pp. 393–408.

[10] P. Heysters, G. Smit, and E. Molenkamp, "A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems," *Journal of Supercomputing*, vol. 26, no. 3, pp. 283–308, Nov. 2003.

[11] H.Amano et al., "MuCCRA Chips: Configurable Dynamically-Reconfigurable Processors," in *Proc. of ASSCC 2007*, Nov. 2007, pp. 384–387.

[12] H.Singh, L.Ming-Hau, L.Guangming, F.J.Kurdahi , N.Bagherzadeh, and E.M.Chaves Filho, Eds., *MorphoSys: an integrated reconfigurable system for data-paralleland computation-intensive applications*. IEEE Transactions on Computers, May 2000, vol. 49, no. 5, pp. 465–481.