A Cost-Effective Context Memory Structure for Dynamically Reconfigurable Processors

Masayasu Suzuki, Yohei Hasegawa, Vu Manh Tuan, Shohei Abe, and Hideharu Amano

Graduate School of Science and Technology, Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama 223-8522 JAPAN
drp@am.ics.keio.ac.jp

Abstract

Multicontext reconfigurable processors can switch its configuration in a single clock cycle by providing a context memory in each of the processing elements. Although these processors have proven to be powerful in many applications, the number of contexts is often not enough.

The context translation table which translates the global instruction pointer, or the global logical context number, into a local physical context number is proposed to realize a larger application while reducing the actual context memories. Our evaluation using NEC Electronics' DRP-1 shows that the proposed method is effective when the size of the tile is small and the number of context is large. In the most efficient case, the required number of contexts is reduced to 25%, and the total amount of configuration data becomes 6.9%.

The template configuration method which extends this idea harnesses the power of multicontext devices by storing basic contexts as templates and combining them to form the actual contexts. While effective in theory, our evaluation shows that the return in adopting such mechanisms in more finer processors as the DRP-1 is minimal where the size of the context memory adds up relative to the number of processing units.

1. Introduction

A chip combining an embedded CPU and a coarse grain dynamically reconfigurable fabric has received attention as a solution to cope with the increasing complexity and development costs brought about by System-on-Chips (SoC). Since the configuration of a coarse grain reconfigurable device is flexible, the same chip can be used for various applications. It can also be "re-fitted" after shipment by rewriting the configuration data. By changing the configuration during execution, the same semiconductor area can be used for various tasks of a single job, thereby improving the area efficiency with time-multiplexed execution.

Some of these devices employ a multicontext structure [9, 15] that provides a set of configuration memory modules in each of the processing elements. By broadcasting the pointer to the individual configuration memories, the hardware configuration can be changed in cycle-by-cycle basis. Hardware configuration is usually referred to as a *hardware context*, and these contexts are interchanged to realize different tasks. Such multicontext dynamically reconfigurable processors execute a single task by changing several hardware contexts, and raises the per-area ratio in performance to ordinary semiconductor chips [3]. Commercial chips have been available [13, 10, 12] and have also been embedded in a portable game engine [11].

As the complexity in the demands placed on these devices increases, there is often a mismatch in the actual number of contexts that is called for, and the physical number of contexts made available. These intense applications include image compression applications that spans over a number of contexts, and error correction code that could only be realized with several dozens of contexts.

In order to cope with the problem, virtual hardware techniques [2] which replace the configuration data in the context memory dynamically during execution from off-chip large scale memory have been researched. The technique, however, is only efficient when the context can be stored in two main iterations of a task, and in reality, a large number of hardware contexts is necessary to keep the device working without stalling.

In multicontext dynamically reconfigurable processors, the context memory is distributed in each of the processing element, and drastically increasing the number of contexts would hinder the per-area efficiency that is otherwise the advantage of these devices.

On the other hand, we have shown in our previous implementations that the number of processing elements used in each of the contexts is different from each other [14]. This means that the context memory is not efficiently utilized in all contexts. In order to efficiently use the context memory, we propose to separate the contexts into logical and the physical contexts. By providing a translation table from the logical context number to the physical one, we show that both the memory requirement and the loading speed of configuration data can be improved.

2. Traditional Methods

A logical diagram of multicontext reconfigurable devices is often drawn as shown in Figure 1. Like common FPGAs, operations of each processing element (PE), connection of components in a PE, and interconnection between PEs are fixed by configuration data stored in configuration or instruction memory. Unlike common reconfigurable devices, multicontext reconfigurable devices provide a set of configuration memory modules which are connected to PEs and interconnects that run between them through a multiplexer. By switching the multiplexer, the datapath can be changed in cycle-by-cycle basis. A *context* is a combination of PEs and the datapath that connects them in a specific order: if PE *A* is connected to PE *B* via the interconnects, this is one context. If PE *A* were then connected onto *C*, we have a brand new context. Multicontext devices store number of contexts within the chip, and switches between them according to the pointer that comes from the central controller. The context memories are usually distributed throughout the chip, with many found as a part of the individual PEs.

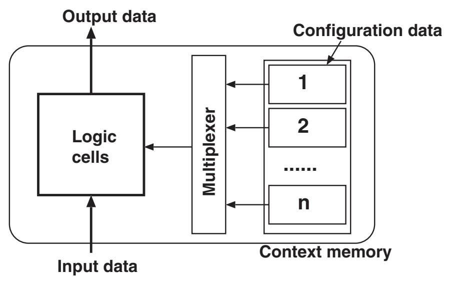

Figure 1. Logical diagram of a multicontext reconfigurable device

The diagram, however, applies only in theory, and most of the multicontext reconfigurable devices are implemented with the mechanisms shown in Figure 2. Each PE provides a memory module that stores the configuration data sets for the corresponding PE and interconnection of surrounding buses. The context number is broadcasted throughout the chip, and used as a pointer to the context memories. By changing the context number and reading the context memory simultaneously, the context is switched in a clock cycle. This in turn means that the configuration data for a context is distributed to each PEs, and the switching of the multiplexer in Figure 1 is replaced by configuration data read-out from each of the context memories.

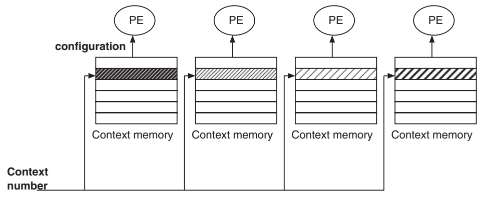

Figure 2. Context switching mechanism used in most multicontext devices

2.1 Target model: NEC Electronics' DRP

Although the configuration memory reduction method proposed here can be applied to any multicontext reconfigurable device using the above context switching structure, we have selected DRP-1 [10] as the target for our study.

2.1.1 DRP-1

DRP is a coarse grain multicontext reconfigurable core which can be integrated into ASICs and SoCs. The primitive unit of DRP core is called a *tile*, and a DRP core consists of arbitrary number of tiles. The number of tiles can be expandable, horizontally and vertically.

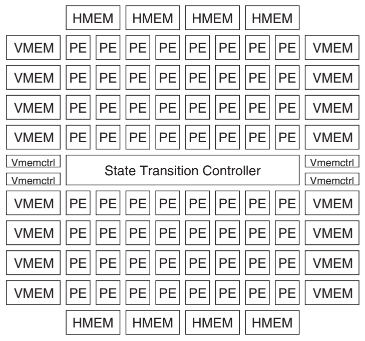

Figure 3. Structure of a tile

The primitive modules of a tile are processing elements (PEs), a State Transition Controller (STC), 2-ported memories (Vertical Memories or VMEMs), its controller (VMCtrl), and 1-ported memories (Horizontal Memories or HMEMs). The structure of a tile is shown in Figure 3.

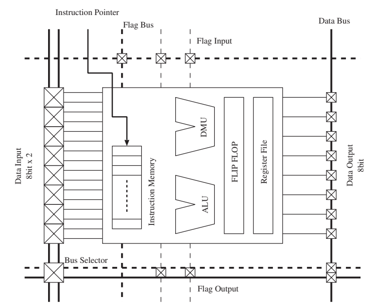

Figure 4. Structure of a PE

There are 64 PEs located in one tile. The architecture of a PE is shown in Figure 4. It has an 8-bit ALU, an 8-bit data management unit (DMU; for shifts/masks), sixteen 8-bit register file units (RFU), and an 8-bit flip-flop (FFU). These units are connected via programmable wires specified by configuration data, and their bit-widths range from 8 bytes through 18 bytes depending on the location. PE has 16-depth context memories and supports multiple context operation. Its context pointer is delivered from the STC.

The STC is a programmable sequencer in which finite state machine (FSM) can be stored. STC has 64 states, and each state is associated with an instruction pointer. FSM of STC operates synchronized with the internal clock, and generates the context pointer for each clock cycle according to the state. Also, STC can receive event signals from PEs to branch conditionally.

As for the memory units, a tile has sixteen 2-ported VMEMs on its right and left sides, and eight 1-ported HMEMs on its upper and lower boundary. The capacity of a VMEM is 8-bit$\times$256-word, and four VMEMs can be handled as a FIFO, using VMCtrl. HMEM is a single-ported memory and it has a larger capacity than the VMEM. It has 8-bit$\times$8K-word entries. Contents of these memories, flip-flops, and register files of PEs are shared by all contexts.

The DRP core, consisting of several tiles, can change its contexts every cycle with the instruction pointer distributed from the central STC (CSTC). The individual STCs within the tiles can also run independently by programming different FSMs. The prototype chip DRP-1 consists of a DRP core with eight tiles. It is fabricated with 0.15-μm 8-metal layer CMOS process. It consists of 8-tile DRP core, eight 32-bit multipliers, an external SRAM controller, a PCI interface, and 256-bit I/Os. The maximum operation frequency is 100-MHz.

An integrated design environment for DRP-1 which includes a high level synthesis tool, a design mapper for DRP, simulators, and a layout/viewer tool is provided. An application program can be written in a C-based high level hardware description language, synthesized, and mapped directly onto the chip.

2.1.2 Context switching model used in DRP-1

Unlike the basic context switching mechanism shown in Figure 2, the DRP-1 changes its context by a built-in sequencer called central state transition controller (STC). The pointer for the context is built in the state transition table provided within the STC as shown in Figure 5. Using this mechanism, finite state machine can be realized with multiple states (in the current DRP-1, four at maximum) per each context. For example, in performing an iteration, only one context is necessary as shown below:

CONTEXT 1: STATE 0: Input data; set variable ITERATION to 1; go to STATE 1

CONTEXT 1: STATE 1: If variable ITERATION is zero, DO task A at CONTEXT 8; else go to STATE 2

CONTEXT 1: STATE 2: If the sum from task A is less than 1000, redo task A by setting 1 to variable ITERATION, and go to STATE 1; else set ITERATION to zero and go to STATE 1

By grouping states that consume very little PEs into a single context, the number of necessary context can be reduced. Even an iteration formed by multiple states can be mapped into a single context if there is enough PEs at the same moment in time.

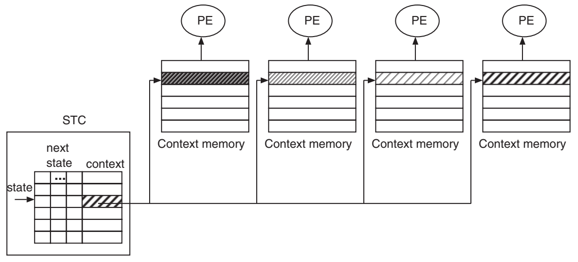

Figure 5. Context switching mechanism used in DRP-1

3. Motivation and Related Work

Stream processing applications usually consist of several tasks which are executed sequentially. Although simple but intensive tasks for processing JPEG2000 and Viterbi decoders have been implemented in the current chip [14], it cannot handle the complicated tasks (e.g. motion vector search) used in recent image processing algorithms such as MPEG-2 and H.264. When the chip size is limited, the usable number of tiles would also be limited. In such cases where only a small number of tiles are available, the number of required contexts becomes large [3]. At the same time, preceding contexts should be stored in the context memories so as not to interrupt operation. In order to satisfy such requirements, the number of contexts would become hundreds or even thousands.

Because the context memory must be distributed in each of the PEs in a multicontext reconfigurable device, the area of the context memory becomes a crucial factor for the PEs. For example, in IMEC's reconfigurable processor ADRES, half of the PE area is occupied by the 32 word configuration memory that are 40-bits each [16]. Large context memory also introduces the overhead to load the configuration data when the device is initialized or when a totally different task is introduced onto the device.

Techniques which reduce the context memory have been researched. Virtual hardware [5, 9] enables to load the configuration data for the next task during execution. Since the order of task execution is fixed, this mechanism can reduce the switching overhead between tasks. However, this mechanism is only useful when a task itself or main iteration can be executed with the number of contexts available on the chip. If the size of the main iteration is beyond a half of the context number which can be stored on the chip, the execution speed is much degraded. Differential configuration [2] is a technique which can reduce the overhead, but it is only applicable when the application has contexts that are similar to each other.

On-the-fly decoding of the compressed configuration data is another approach in reducing the configuration loading time [8]. This approach, however, is not so efficient for coarse grain devices when compared with fine grain devices like common FPGAs [7], and Kitaoka and his colleagues only reduced the data size by 20%-40%.

4. Context Translation Table

4.1 Logical and physical context numbers

The global pointer method shown in Figure 2 and Figure 5 accesses the same address of every context memory. This means that the context memory is accessed even if the PE is not used in the context. Through the implementation of several applications, we have learned that it is very difficult to equal out the number of PEs in all of the contexts [14]. The PE usage is different in each context, and the utilization of context memory is sporadic.

In order to manage the context number independently by each PE, a table is provided in each PE to translate the global logical context number into a local physical context number. As shown in Figure 6, the physical context number can be assigned independently by each PE, and all contexts that are not utilized are represented as "NULL". NULL entries can be used for storing configuration data of other contexts. The table for the translation is called the context translation table. The same mechanism can be used when multiple states are assigned into a context as in the DRP-1.

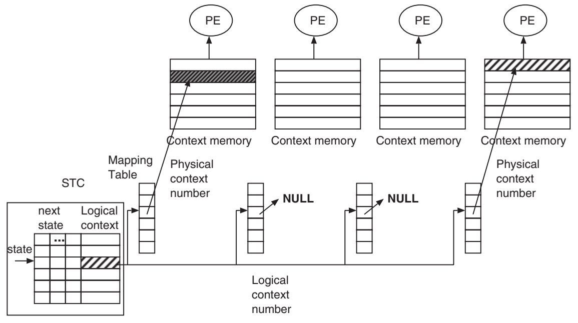

Figure 6. Context number translation tables

There are following two disadvantages in this scheme:

- The cost for translation table is added, and
- There is overhead in reading out the physical context number.

The former is the trade-off between increasing memory requirement for the translation table and decreasing context memory, and we will evaluate this in the next section. The latter stretches the operational clock period if the FSM spans over multiple states and are complicated. The conditions for state transition, however, are quite simple in applications implemented so far [14], and there were no cases where the delay time for number translation hindered the total system.

4.2 Sharing physical local contexts

By managing local context number in each PEs, the same configuration data can be shared by multiple global logical contexts. This means that the PE is used for the same purpose in multiple contexts, and the configuration data can be shared and reused just by setting the same physical context number in the translation table. Figure 6 also shows the sharing method. Since the same operations and data transfer tend to be used in multiple contexts in the same task, this technique can reduce the total amount of context memory. We call this technique the *physical context sharing method*.

4.3 Evaluation based on real applications

4.3.1 Applications

The possibility of memory reduction by introducing the context translation table is evaluated by using actual stream applications (Table 1) implemented on the DRP-1. We show both cases where a single tile or all eight tiles are used to implement the DCT and IMDCT. Designs which require more than 16 contexts cannot be executed directly in the current DRP-1 chip, while smaller designs can be tested on the actual chip.

Table 1. Tested applications

DCT	Discrete Cosine Trans. in JPEG encoder	[14]
CVT	Converter in JPEG encoder	[14]
IMDCT	Inverse Modified DCT in MP3 decoder	[17]
DWT	Wavelet transform in JPEG2000 encoder	[4]
MQC	Arithmetic coder in JPEG2000 encoder	[4]
AES	Encryption System using Rijndael	[1]

4.4 The maximum number of required physical contexts

By providing the context translation table, the required number of physical context varies according to the PEs. From the viewpoint of reducing context memory, the maximum number of required contexts is important. We therefore show the case without the context translation table (CTX_{org}), the maximum number of required contexts when it is introduced (CTX_D), and the case when physical contexts are shared (CTX_{DS}) in Table 2.

Table 2. Reduction of context numbers

Appl.	Tile	CTX_{org}	CTX_D	CTX_{DS}	TOP_n
DCT	1	70	36	18	7
DCT	8	16	16	13	22
CVT	2	28	17	9	8
IMDCT	1	70	40	27	9
IMDCT	8	16	16	11	13
MQC	1	10	10	8	15
AES	1	7	7	5	11

This table shows that the context translation table is effective when the number of tiles is small and the number of required context is large. For example, the required context number becomes half of the original just by introducing the context translation table in DCT implemented with one tile, and it can further be reduced by sharing. On the other hand, in cases where the context numbers are small, there is less return. However, by using physical context sharing, the required contexts become 68%-80% of the original implementation.

Since the context translation table itself requires memory, the total memory requirement including context memory and table must be evaluated. The translation table needs entries corresponding to the logical context number and the bit width for physical context number. Here, the total memory amount (bits) with the table is represented with the following expression:

$$CTX_{org} \times \lceil log_2 CTX_{DS} \rceil + n \times CTX_{DS}$$

where n is the number of configuration data bits for each PE.

On the other hand, the original method without translation table requires:

$$n \times CTX_{org}.$$

n varies depending on the PE architecture, and usually runs from 50-bits to 120-bits[1]. TOP_n shown in Table 2 shows the value of n when both of the total memory requirements are the same. If n is greater than TOP_n, using the context translation table is better than going without it. Since the largest value of TOP_n is 22 and most PE would require 50-bits for its configuration, employing the context translation table returns more contexts in all applications.

4.5 Total configuration data

The proposed method can reduce the total amount of configuration data by sharing the same physical contexts. By reducing the configuration data, the time to load the configuration data can also be reduced. Table 3 shows the ratio (R) of required configuration data by the proposed method to one by the original method. Note that the data stored in the translation table is included. The tendency is almost the same as Table 2 which shows the maximum number with some exceptions, but the total amount of configuration data becomes 6.9%-42.7%. This result is better than Kitaoka's compression method which reduces 50% at maximum [7]. We therefore conclude that the proposed method has an advantage in reducing the loading time of configuration data.

Table 3. The ratio of total configuration data

Appl.	# of Tile	R(%)
DCT	1	12.8
DCT	8	13.8
CVT	2	35.0
IMDCT	1	6.9
IMDCT	8	16.4
AES	1	42.7
MQC	1	30.0

[1] n for DRP-1 has not been made public.

5. Template Configuration

5.1 Concept of template configuration

The context translation table only enables to share physical contexts with the completely same structure, and similar contexts with a minor difference cannot be easily handled. Because there is so many possibilities in the configuration of a PE, there may be cases where all but one of the component is different while others are completely the same. An example of this may be a case where all the operations of the PE between two contexts are the same except for the address of the output register.

In order to cope with such cases, template configuration method is proposed. As shown in Figure 7, a portion of the frequently changing configuration data is defined as a variable part called the *modifier*, and the remaining common part is called the *template*. The template is accessed with the local physical address, while the modifier is stored in the context translation table and indexed by the global logical number. The configuration data is generated from the summation of the template data with the modifier. This scheme, called the template configuration, is advantageous when an application consists of common but slightly different contexts.

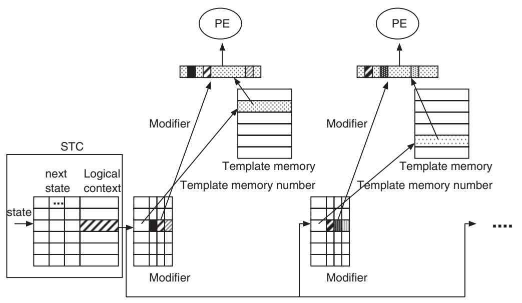

Figure 7. Template configuration

The problems associated with template configuration are as follows:

- Since the modifier is stored in the translation table, there is increase in the required memory.
- It is difficult to decide which part should be selected as the modifier.
- If the modifier includes the immediate data or configuration data for interconnection, the bit length tends to be large.

If this idea were to be compared against the instruction set of common computers, the physical address indicating the template can be considered as the operation code, and the modifier corresponds to the operands or immediate data. From this point of view, the template configuration is somehow similar to the vertical micro-instruction code used in old micro-programmable computers. The major difference between them is that template configuration is used in each of the PE in the array.

The concept of template configuration is similar to the differential configuration [2]. The difference exists in that the pairs of address and the configuration data for each differential part is stored for each context in differential configuration. In template configuration, the template is provided for each PE, and the location of the modifier is fixed.

5.2 Similarity of contexts

Template configuration is only effective when a template can be shared with a certain number of contexts. This means that the configuration data for a PE differs only slightly in a certain number of contexts. We therefore investigated the similarity of configuration data for a PE that requires the largest context numbers in each application as shown in Table 1. Table 4 shows the number of contexts whose configuration is slightly different from others.

Table 4. Similarity of PE structure in all contexts

Application	# of. similar contexts	Different part
DCT (1Tile)	2	Connection
	2	Register usage
	2	DMU operation
DCT (8Tile)	2	DMU operation
	2	DMU operation
CVT	2	Register usage
IMDCT (1Tile)	2	Register usage
	2	Register usage
IMDCT (8Tile)	2	DMU operation
AES-EBC	2	DMU operation
	2	ALU operation
	2	Register usage
DES	2	DMU operation
	2	ALU operation

From this table, we conclude that there are PEs in every application that are similar but partially different. We also learned that there were not too many of such cases. In this investigation, there are no cases where a PE took a similar structure in more than two contexts. Furthermore, the different part varies depending on the application.

We also investigated the number of contexts which can be reduced by sharing contexts with template configuration. Table 5 shows the relative number of contexts to the total contexts used for each PE. *Match* is the number of completely same contexts that can be reduced with the physical context sharing proposed in the previous section. *DMU* is the reduced number when the function of the DMU is chosen as the modifier. Other columns *ALU*, *RFU*, *FF*, and *SW* show cases where the corresponding part is chosen as the modifier. From Table 5, DMU has an edge over the other modifiers except in CVT and AES-EBC. The relative number is not large compared with the number of completely same contexts.

Table 5. The ratio of context number which can be reduced

Appl.	Match	DMU	ALU	RFU	FF	SW
DCT (2T)	0.43	0.010	0.025	0.019	0.014	0.003
DCT (4T)	0.25	0.13	0.044	0.012	0.003	0.0
DCT (6T)	0.20	0.12	0.023	0.025	0.004	0.003
DCT (8T)	0.12	0.11	0.027	0.026	0.003	0.004
CVT	0.27	0.015	0.028	0.018	0.006	0.005
IMDCT	0.11	0.047	0.041	0.016	0.003	0.005
DES	0.19	0.085	0.023	0.043	0.0	0.001
SHA1	0.21	0.010	0.041	0.0	0.0	0.0
AES-EBC	0.20	0.003	0.015	0.025	0.0	0.0

From these tables, we have no choice but to conclude that the effect of template configuration is small when it is applied to DRP-1. This comes from rather small granularity of PEs (8-bits) in DRP-1, and we will try this method for other coarser grained architectures.

Another method to extend the template configuration is to share a template by multiple PEs. Figure 8 shows this concept. By sharing the template memory with multiple PEs, the similarity between PEs can be used. However, this extension accompanies the access conflict problem, and the delay for transferring the template to each PE will be another problem. The extension of the concept of template to the horizontal direction is our future work.

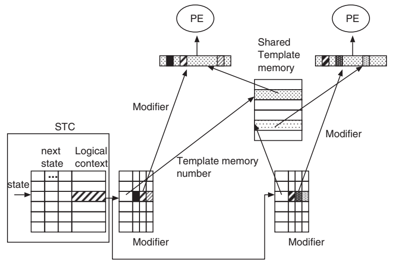

Figure 8. Sharing template memory

6. Conclusion

The context translation table which translates a global logical context number into a local physical context number is proposed to reduce the context memory and to raise the total amount of configuration realizable on a multicontext reconfigurable device. From the evaluation using NEC Electronics' DRP-1, it appears that the proposed method is worthwhile when the size of the tile is small and the number of context is large. In the most efficient case, the required context number is reduced to 25%, and the total amount of configuration data becomes 6.9%.

Template configuration which bundles similar contexts to return more contexts is also proposed. Our evaluation results show that such idea cannot be used effectively in a device like DRP-1 where the PEs are small in size. Our future work is to apply the method to a system with a more coarser grained PE.

Acknowledgments

The authors would like to thank NEC Electronics and the DRP Group at NEC Laboratories for their valuable advice. A part of the simulation used in this project was supported by the Mentor Graphics University Program.

References

[1] S. Abe, Y. Hasegawa, and H. Amano. Implementation of AES on the Dynamic Reconfigurable Processor. In *Proc. of IEEE Cool Chips VIII*, pp.192, 2005.

[2] H. Amano, T. Inuo, H. Kami, T. Fujii, and M. Suzuki. Techniques for Virtual Hardware on a Dynamically Reconfigurable Processor. In *Proc. of FPL*, pp.464–473, 2004.

[3] H. Amano, et al. Performance and Cost Analysis of Time-multiplexed Execution on the Dynamically Reconfigurable Processor. In *Proc. of FCCM*, 2005.

[4] K. Deguchi, et al. Implementing Core Tasks of JPEG2000 Encoder on the Dynamically Reconfigurable Processor. In *Proc. of International Workshop on Dynamically Reconfigurable Systems*, March 2005.

[5] R. Enzler, C. Plessl, and M. Platzner. Virtualzing Hardware with Multi-context Reconfigurable Arrays. In *Proc. of FPL*, pp.151–160, 2003.

[6] F. Furtek, E. Hogenauer, and J. Scheuermann. Interconnecting Heterogeneous Nodes in an Adaptive Computing Machine. In *Proc. of FPL*, pp.125–134, 2004.

[7] T. Kitaoka, H. Amano, and K. Anjo. Reducing the Configuration Loading Time of a Coarse Grain Multicontext Reconfigurable Device. In *Proc. of FPL*, pp.171–180, Sept. 2003.

[8] Z. Li and S. Hauck. Configuration Compression for Virtex FPGA. In *Proc. of FCCM*, pp.142–154, 2001.

[9] X.-P. Ling and H. Amano. WASMII: A Data Driven Computer on a Virtual Hardware. In *Proc. FCCM*, pp. 33–42, 1993.

[10] M. Motomura. A Dynamically Reconfigurable Processor Architecture. Microprocessor Forum, 2002.

[11] M. Okabe, et al. An 90nm Embedded DRAM Single Chip LSI with a 3D Graphics, H.264 Codec Engine, and a Reconfigurable Processor. Hot Chips, 2004.

[12] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner. The XPP Architecture and Its Co-simulation within the Simulink Environment. In *Proc. of FPL*, pp.761–770, 2004.

[13] T. Sugawara, K. Ide, and T. Sato. Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology. *IEICE Trans. on Inf.&Syst.*, Vol.E87-D, No.8, pp.1997–2003, May 2004.

[14] M. Suzuki, et al. Stream Applications on the Dynamically Reconfigurable Processor. In *Proc. of ICFPT*, Dec. 2004.

[15] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *Proc. of FCCM*, pp. 22–28, 1997.

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

[17] Y. Yamada, et al. Core Processor/Multicontext Device Codesign. In *Proc. of Cool Chips VI*, pp.82, 2003.