A Retargetable Compiler for Cell-Array-Based Self-Reconfigurable Architecture

Masayuki Hiromoto†, Shin'ichi Kouyama†, Hiroyuki Ochi†, and Yukihiro Nakamura††

† Kyoto University, Yoshida-honmachi, Sakyo-ku, Kyoto, 606-8501,Japan
†† Ritsumeikan University, 1-1-1, Noji-Higashi, Kusatsu, Shiga, 525-8577, Japan

Summary
Simulation-based quantitative performance evaluation using specific applications is indispensable for developing architectures of self-reconfigurable devices since static analysis is difficult to estimate their performance. In order to generate configuration data needed for simulating various target architectures, we developed a synthesis tool which can be retargeted to various self-reconfigurable devices specified by architecture parameters. Given an application in C-language, our tool automatically executes data-flow analysis, technology mapping, and layout synthesis. Our tool enables us to perform efficient design-space exploration, and its retargetability helps fair evaluation of the devices on the same platform. This paper also shows architecture evaluation examples using our tool to demonstrate the advantage of our tool.

Key words:
coarse-grain, ALU-based reconfigurable architecture, high-level synthesis, layout synthesis

1. Introduction

Recently, dynamic reconfigurable devices have been remarkably developed. Dynamic reconfigurable devices have flexibility in changing its functionality even in runtime, while functionality of Application Specific Integrated Circuits (ASICs) cannot be changed after fabrication. Dynamic reconfigurable devices achieve higher performance than processors using potential parallelism in the application. Self-reconfigurable devices such as PCA [7, 14] are a special class of dynamic reconfigurable devices each of whose basic cell can be reconfigured individually by its own decision. The device has a uniform array structure of basic cells, which work independently to realize distributed processing. These self-reconfigurable devices are supposed to achieve high performance and flexibility and thus they are expected to be useful for today's portable devices which support several functionalities and/or standards.

In our research, we explore an architecture of reconfigurable device featuring self-reconfiguration and distributed processing through evaluation and comparison of device architectures. Performance of self-reconfigurable devices, however, is difficult to be estimated only with static analysis; therefore simulation-based quantitative evaluation using specific applications is indispensable. Unfortunately, simulation-based scheme requires configuration data of the target application dedicated for the architecture under evaluation. Generation of configuration data costs large man-hours without any development tools dedicated for the architecture. In this paper we propose a C-compiler, which automatically generates configuration data of an application, to support exploration of device architectures.

Note that retargetability is important for a compiler or a synthesis tool used for architecture exploration. Since most recent compiler or synthesis tools for reconfigurable devices, such as FPGAs, are developed and targeted only for a specific architecture, they do not suit for a use of developing new reconfigurable architecture. For such use, VPR [2], which is a placement and routing tool for FPGAs, is available for architecture exploration since it can be used for most island-style FPGAs. However, there are no tools widely applicable for developing self-reconfigurable architectures. Therefore, we developed a retargetable compiler for self-reconfigurable architectures.

In addition to introducing our retargetable compiler, we also show architecture evaluation examples using the compiler on some applications to demonstrate the advantage of our tool.

This paper first describes about self-reconfigurable devices in Section 2, and introduces our evaluation platform for self-reconfigurable devices in Section 3. Then detailed explanations of proposed compiler are described in Section 4, and architecture evaluation with proposed compiler is shown in Section 5. Finally, a conclusion is described in Section 6.

2. Self-Reconfigurable Device

2.1 Target Devices in this Paper

In this paper, we define a self-reconfigurable device, the target architecture of our compiler, as a class of dynamic

Manuscript received April 25, 2007
Manuscript revised April 25, 2007

reconfigurable architecture that has following two features:

1. The device consists of a uniform array of basic cells.
2. Each basic cell consists of a reconfiguration controller and a reconfigurable resource so that reconfiguration can be triggered and completed locally.

1. provides scalability of the device. 1. also enables us to allocate application tasks at arbitrary place (relocatability). 2. is indispensable to realize parallel/distributed control scheme to avoid bottlenecks and overhead caused by a central controller.

2.2 PCA

This part introduces one of the self-reconfigurable devices, PCA [7, 14]. PCA has a uniform array structure of basic elements called "PCA Cells", which includes "Plastic Parts" and "Built-in Part" respectively (See Figure 1). The plastic part consists of Look-Up-Tables (LUTs) like FPGAs and reconfigured to arbitrary logic circuits. The built-in part controls data flow and reconfiguration of the plastic parts inside or outside of the cell. The built-in part accepts commands from the accompanied plastic part or other cells. Each PCA cell is connected to the neighbor cells and this provides scalability of a PCA device. With self-reconfiguration, PCA can create, copy, or delete circuit modules to perform flexible and adaptive processing.

Several PCA devices are proposed such as PCA-1 [9], PCA-2 [6], and PCA-Chip2 [18], which are all LUT-based fine-grained reconfigurable devices. Although fine-grained scheme achieves high flexibility, it requires large amount of configuration data and long time to reconfigure the devices. Since large configuration data and long reconfiguration time become significant overheads for dynamic reconfigurable devices, most recent dynamic reconfigurable devices such as DAPDNA [17] and DRP [13] adopt coarse-grained structure to improve performance. We expect that similar improvement can be made for self-reconfigurable devices by adopting coarse-grained structure.

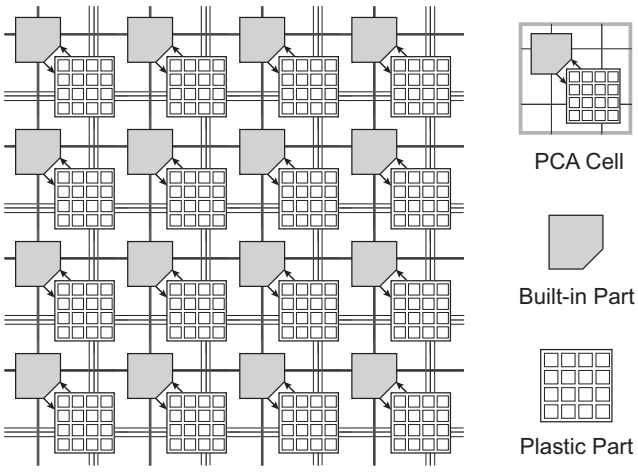

Fig. 1 Array structure of PCA

3. Evaluation Platform for Self-Reconfigurable Devices

Our colleagues have proposed an evaluation platform for self-reconfigurable devices [10] to explore self-reconfigurable architectures. This section introduces the platform and the role of the compiler proposed in this paper.

3.1 Target Architecture

The target architecture of the simulation platform is those defined in Section 2.1. The platform enables us to evaluate and compare the performance of various architectures quantitatively by simply modifying parameters. The possible architecture parameters include reconfigurable resource in each basic cell (ALUs with various input word length and supported operations), wire resources in the array (structure and bandwidth), and mechanism for configuration delivery.

3.2 Overview of Platform

An overview of the platform is shown in Figure 2. The platform consists of device generator, compiler, and a shared library. An architecture designer can use this platform in conjunction with general RTL simulators and HDL synthesis tools.

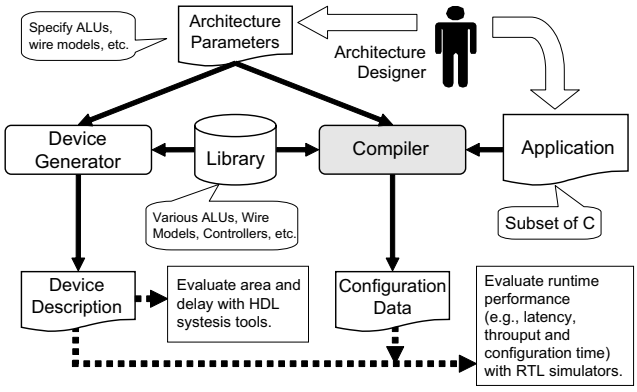

Fig. 2 Evaluation platform for self-reconfigurable devices

Using the platform, architecture evaluation is performed as follows; first a designer determines parameters of an architecture to be evaluated and input them to the platform. Then the platform selects parts and resources from the shared library according to the given parameters, and generates a device description for the desired architecture. Since this is described by Verilog HDL, which is one of the widely used Hardware Description Languages (HDL), the designer can estimate static features of the device, such as circuit area and wire delay, for specific process technology by using general RTL synthesis tools.

To evaluate the dynamic nature of the device such as latency, throughput, and reconfiguration time for specific application, the designer proceeds to simulation-based experiments. In our platform, the designer can simulate the device behavior with the device description and the configuration data of applications by using general RTL simulators. Note that simulation-based performance estimation requires configuration data of applications to be executed on the target device.

3.3 Compiler's Role

As described above, generating configuration data is indispensable to evaluate run-time performance of the target device using simulation with applications. However, since the platform in [10] does not include an automatic generator of configuration data from application description, this process takes long time and large man-hours. To improve efficiency of architecture exploration, a compiler that automatically generates configuration data is desired.

To enable description of applications easy, it is desired to support high-level language (such as C) for source code of compilation. Another requirement is retargetability, which makes the compiler usable for various kinds of architectures. A retargetable compiler allows quantitative evaluation between different architectures on the same platform.

4. Proposed Compiler

4.1 Overview

Figure 3 shows a processing flow of proposed compiler. Input of the compiler is simplified C language that does not support all the C grammar. The compiler first converts a given C code to "GCC Tree" expression using a front-end of GNU Compiler Collection version 4.0 (GCC-4.0). The GCC Tree, which is intermediate expression used inside GCC, represents syntax trees of an input C code and architecture-independent optimizations are performed on the GCC Tree. Next, the compiler generates a Data Flow Graph (DFG), which represents a flow and dependency of data and control, from the GCC Tree. Then nodes in the DFG are assigned to ALUs to generate a netlist of ALUs. Finally, all the ALUs are placed and routed according to the netlist and configuration data is generated. Details of each part are described in the following subsections.

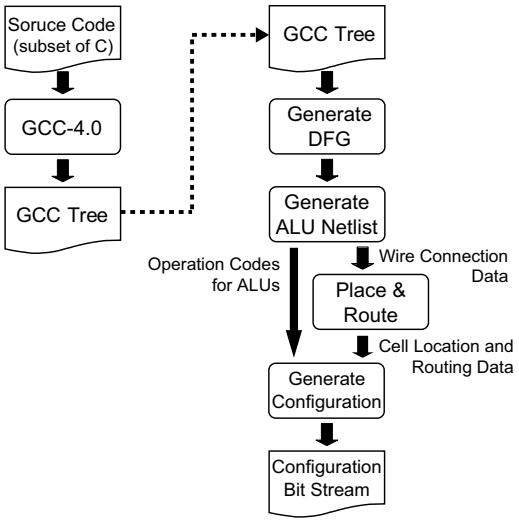

Fig. 3 Overview of compile flow

4.2 GCC Tree Generation

First, the compiler needs to parse C programs because an input of the compiler is written in C language. C programs usually contain redundant description and elimination of such redundancy improves processing performance. Therefore, our compiler adopts GCC as a front-end to perform optimization on input source codes. Major optimizations that GCC performs are Tree SSA scheme [15, 16] based-on Static Single Assignment form (SSA)

[4], constant folding, algebraic simplification, common subexpression elimination, and so on.

4.3 DFG Conversion

The Tree generated by GCC shows a control and data flow suitable for sequential execution on a processor. Our compiler converts the GCC Tree to a DFG that is suitable for parallel processing with hardware implementation. A DFG, which consists of a set of nodes and arcs, is generated to include all the information of the input source code. The nodes and the arcs are associated with data or operation and directions of data flows, respectively. The conversion of a GCC tree to a DFG is not easy, because processing methods of a hardware is different from those of a processor. Especially, a conversion of conditional branches with if and else statements are introduced in this part.

Figure 4 (a) shows a conditional branch for execution on processors. First a condition statement is evaluated and then one of two paths is selected according to the evaluation result. For hardware implementation, however, a DFG is better to be expressed like Figure 4 (b). While evaluating a condition, all the possible paths can be executed in parallel at the same time. After parallel execution, a result is selected according to the evaluated condition. Since GCC Tree gives representations like Figure 4 (a), the proposed compiler converts conditional statements to the DFG like Figure 4 (b).

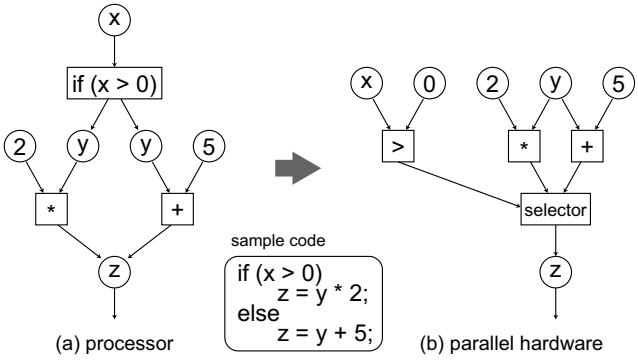

Fig. 4 A sample of a DFG which contains a condition branch

4.4 ALU Netlist Generation

This part of the compiler generates a netlist of the application which consists of ALUs and wire. This netlist is generated by mapping each operation node in the DFG to ALU(s) on a target device. While the compile processes described so far are independent of a target architecture, the rest of the compile processes including this ALU mapping phase depends on the architecture parameters given by the designer.

A procedure of generating a netlist is as follows; first search operation nodes one-by-one in the DFG from the root of the tree, and insert ALU(s) associated to each node to a netlist.

A DFG node can be mapped to an ALU in a straight-forward manner, if the operation performed in the node is covered by the instruction set of the ALU of target architecture. If the required word length is larger than that of the ALU, multiple ALUs are associated to expand the word length. If the required operation is not supported by the ALU (*e.g.*, multiplication is used in the application, but the ALU does not support multiplication,) the compiler picks up a multi-ALU unit for the operation from the library.

4.5 Placement

The compiler determines the physical location for each ALU in the generated netlist on the cell array of the target self-reconfigurable device. We adopt a pairwise exchange method using a simulated annealing [8] algorithm, which is often used for layout algorithm of LSI design, as an ALU placement method. In a pairwise exchange method, beginning with a random initial layout, randomly selected two ALUs are exchanged so as to improve a cost function that indicates whole layout goodness. The most important problem for the simulated annealing scheme is a definition of the performance function. Since the cont function that is calculated iteratively requires a low computational cost and adequately expressing the layout goodness. In this paper, we use total wire length and wire complexity as a cost function.

4.6 Routing

After placement of ALUs, routing wires between ALUs is executed. When routing wires, all routes are desired to have paths as short as possible, but optimum routing is not always accomplished because of limitation of available wire resources.

Among many routing algorithms proposed previously, we adopt Lee's maze routing algorithm [11], which is a simple algorithm and ensures to solve the problems as long as there is a solution. This maze routing method generally takes large computational costs and tremendously long time for a large problem, but a routing problem in this paper is not so complex because our target is ALU-based coarse-grained architecture. Therefore, the maze algorithm is applicable to our proposed compiler.

4.7 Configuration Data Generation

Finally, the compiler generates a bit stream of configuration data that can be downloaded to the target device. The instruction code for each ALU and placement and routing results are converted to associated bit sequence and written on an output file.

5. Architecture Evaluation with Proposed Compiler

In this paper, some applications are implemented on a self-reconfigurable device using the proposed compiler, and performance of the several variations of architecture is evaluated and compared. This section describes the evaluation results, and discusses on the relationship between architecture and applications.

5.1 Preparation

As shown in Section 3, our target architecture for a self-reconfigurable device to be evaluated has a cell-array-based structure. Figure 5 shows an overview of a basic cell used in the experiments. It has an ALU and a register surrounded by wires and multiplexers. Each basic cell is connected to four-direction neighbor cells via multiplexers located in four side of the basic cell. On this framework of the architecture, proposed compiler can map applications to the device according to given parameters. In this paper three demonstrations are performed: evaluation of bit width, wire models, and ALU instruction sets. These demonstrations are shown in the following subsections.

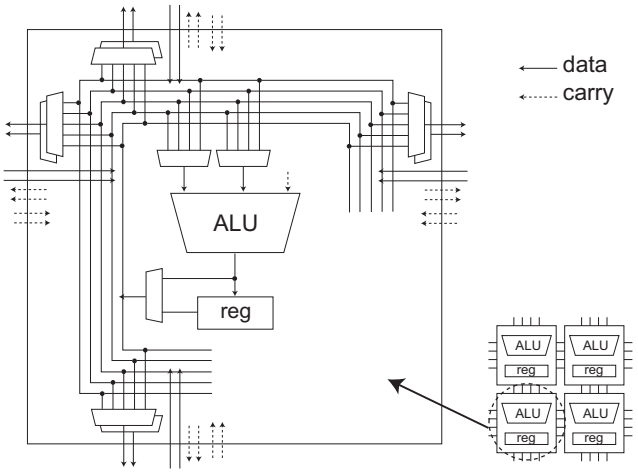

Fig. 5 A structure of a basic cell

We choose total circuit area and configuration data size to evaluate architecture performance. Both of them are significant parameters because circuit area affects whole system size and power consumption, and configuration data size determines reconfiguration time.

5.2 Evaluation of Bit Width

The first demonstration introduced in this section is evaluation of bit width of an ALU. An optimum ALU bit width may differ according to an application executed on the device. To find some relationships between optimum ALU bit width and applications, performance estimation for three architectures with different ALU bit widths (4-bit, 8-bit, 16-bit) are made. As the target applications, we used Discrete Cosine Transform (DCT) in 5.2.1, "SubBytes" process in Advanced Encryption Standard (AES) [1] in 5.2.2, and a decoder of Error Correction Code (ECC) [5] in 5.2.3. DCT is selected as an application including mathematical operations, AES as byte-wise application, and ECC as bit-wise application.

5.2.1 DCT

Algorithm Overview We adopt Chen's fast DCT algorithm [3] and Figure 6 shows a sample block diagram of 8-input 1-dimensional DCT. A part of the circuit rounded by broken line is implemented on the reconfigurable cell array with different bit width ALUs. As shown in Figure 6, this application mainly consists of mathematical operation like multiplication and addition. Note that the input bit width is 8 bits.

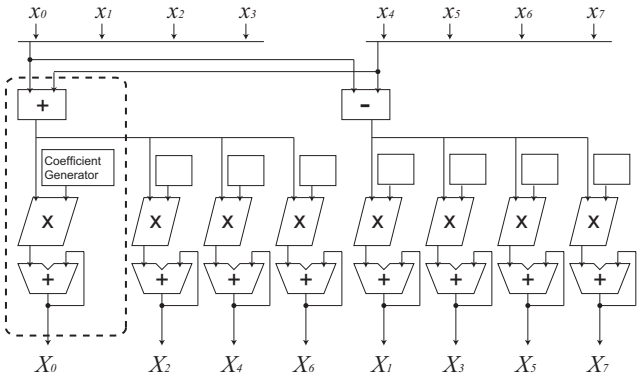

Fig. 6 Block diagram of DCT

Results Compilation results of DCT are shown in Table 1. The first row shows bit width of ALUs and the second shows how many cells are used to realize DCT application. The total area is the product of the circuit area size of a single cell and the number of utilized cells. The area is measured by NAND2-equivalent gate count. The total configuration data is also calculated from number of cells and configuration data bits per cell.

Figure 7 shows the circuit area and configuration data against different bit width, 4-bit, 8-bit, and 16-bit. The 4-bit ALU achieves the smallest circuit size and the 16-bit ALU shows the shortest configuration data. This shows the fine-grained 4-bit ALU is suitable for small circuit use, because fine-grained elements realize high density and efficient layouts. On the other hand, fine-grained structure requires more ALUs than coarse-grained one to realize an identically application. Therefore, for DCT application, the coarse-grained 16-bit ALUs are superior to the others in terms of total configuration data size although a circuit size and configuration data for a single 16-bit cell is larger than the others. This shows that coarse-grained scheme is really effective to reduce configuration data for applications like DCT.

Table 1: Compilation results for DCT

ALU bit width	4-bit	8-bit	16-bit
number of cells	105	63	42
area of a cell	1,277	2,159	3,722
total area	134,085	136,017	156,324
configuration data for a cell	87	91	99
total configuration data	9,135	5,733	4,158

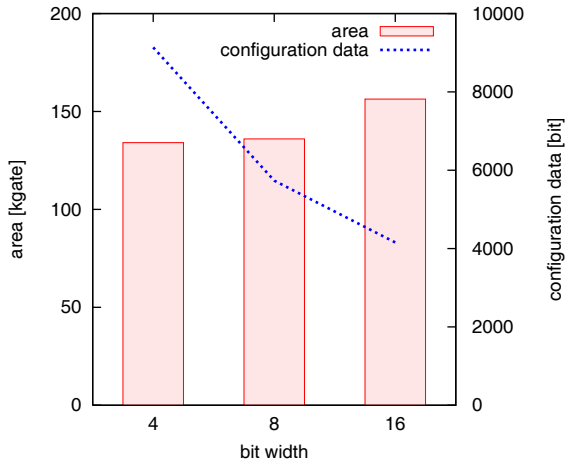

Fig. 7 Compilation results for DCT

5.2.2 AES SubBytes

Algorithm Overview AES is a major encryption standard, which consists of roughly four parts, AddRoundKey, MixColumns, ShiftRows, and SubBytes. Since AddRoundKey and ShiftRows parts can be realized by simple circuits, we evaluate MixColumns and SubBytes processes on target reconfigurable devices. Both of them consist of byte-wise operations, thus we introduce only a result of SubBytes in this paper.

In SubBytes process, a non-linear transformation called "S-Box" is performed for input data of 8-bit width. Output is also 8 bits. As an implementation of S-Box, we used a method based on operations over a composite Galois field $GF(((2^2)^2)^2)$ [12]. This process consists of logical operations and byte-wise array operations.

Results Compilation results are shown is Table 2 and Figure 8. The meaning of each row of the table is similar to Table 1. The results show circuit that area and configuration data becomes smallest when 4-bit and 8-bit ALUs are used, respectively. This is caused by the following reasons. The SubBytes process consists of random logic operations and byte-width operations. Since the random logic operation consumes one ALU for 1-bit operation, fine-grained ALUs are advantageous. On the other hand, 8-bit ALUs are optimum for byte-wise operations because of the identical bit width of ALUs and operations. For SubBytes process, random logic operations are dominant factor for circuit area, while byte-wise operations are dominant factor for configuration data.

Table 2: Compilation results for AES SubBytes

ALU bit width	4-bit	8-bit	16-bit
number of cells	324	228	228
area of a cell	1,277	2,159	3,722
total area	413,748	492,252	848,616
configuration data for a cell	87	91	99
total configuration data	28,188	20,748	22,572

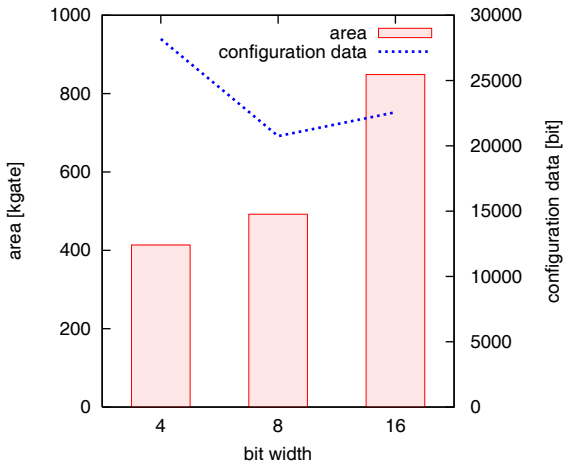

Fig. 8 Compilation results for AES SubBytes

5.2.3 ECC Decoder

Algorithm Overview In this paper we use an ECC that is capable of single-error-correction and double-error-

detection. The code has 16 data bits and 6 parity bits. In a decoding process, 6 syndrome bits are calculated from 22 input bits (16 data bits and 6 parity bits) by bit-wise Exclusive-OR operations. Then the syndrome bits are examined to detect or correct errors by operations on 6-bit data.

Results Table 3 and Figure 9 show compiling results. The meaning of each row of the table is also similar to Table 1. This ECC decoder includes mainly 16-bit operations for data bits and 6-bit operations for parity bits. Although these are bit-wise operations, multiple bits can be processed by a single ALU since our ALU (similar to those in conventional CPUs) supports the bit-wise operation which processes the identical logic operation to all bits in parallel. Therefore, the smallest number of cells is achieved by an architecture with 16-bit ALUs that can execute many bits simultaneously, and 1.5 times of the cells are used with 8-bit ALUs and twice with 4-bit ALUs. In spite of many cells being used in fine-grained design, the total circuit area takes almost identical value in each design because a fine-grained ALU is smaller than coarse-grained one. However, number of used cells directly affects total amount of configuration data, and 16-bit ALUs achieve the smallest value in this case.

Table 3: Compilation results for ECC decoder

ALU bit width	4-bit	8-bit	16-bit
number of cells	545	277	197
area of a cell	1,277	2,159	3,722
total area	697,242	602,361	736,956
configuration data for a cell	87	91	99
total configuration data	47,502	25,389	19,602

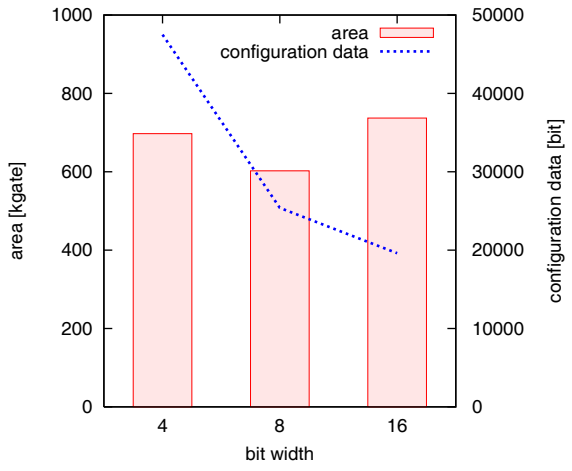

Fig. 9 Compilation results for ECC decoder

These results show that coarse-grained ALUs are also suitable for applications that mainly consist of bit-wise operations if the identical logic operations are executed in parallel.

5.3 Evaluation of Wire Models

As the second demonstration scenario, we will compare architectures with different wire resources using our compiler. An architecture with rich wire resources can implement applications efficiently but physical area for wire resources becomes large. In contrast, an architecture with poor wires cannot achieve high utilization of cells, because wire congestion should be reduced and/or need cells used for “feed-through” purpose. In this section, experimental results of different bit width of wire resources between basic cells is shown in order to demonstrate that the proposed compiler routes paths over the wires of parameterized bit width. The implemented application is DCT, which was described in previous section.

Figure 10 shows the compilation result with interconnection wires of 1 or 2 track(s). “1 track” means that there are wire resources as shown in Figure 5, where each data bus has the same bit width as ALUs. “2 tracks” means that the number of data bus resources is twice as many as Figure 5 to increase the flexibility of routing. Although there is little difference in circuit area between 1 track and 2 tracks cases with ALUs of 4-bit word length, the 1 track architecture achieves 50% reduction from 2 tracks architecture in area for 16-bit ALUs. In addition, the amount of configuration data for 1 track design is less than 2 tracks one at every ALU bit width. Therefore, a single track wire resource is sufficient for implementing applications like a DCT, which does not have very complex interconnections.

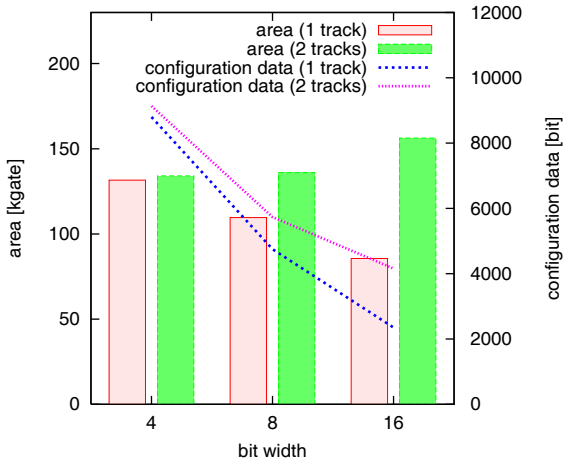

Fig. 10 Compilation results for DCT with different wire tracks

5.4 Evaluation of Instruction Sets

The final demonstration is evaluation of architectures with ALUs of different instruction sets. Since there are many possible variations of ALUs, our compiler can generate ALU netlists according to a given instruction set. During the design of optimum instruction set of ALUs to implement the target applications efficiently, the decision making whether the ALU should have a multiplication instruction or not is very important. We test two instruction sets; one includes multiplication and the other does not, referred to as “with MUL” and “w/o MUL”, respectively. Again, DCT is used for the application.

The compilation results are shown in Figure 11. Since a DCT contains many multiplications, “with MUL” is much better than “w/o MUL” in terms of area and configuration data. By adding multipliers to ALUs, the circuit area decreases to 40% of the design on the "no MUL" architecture although area for a single cell increases from 2,037 gates to 2,857 gates. The configuration data also decreases to 30%.

These results suggest that instruction set of ALUs should be carefully designed for specific target applications, and our compiler is useful for exploring the design space.

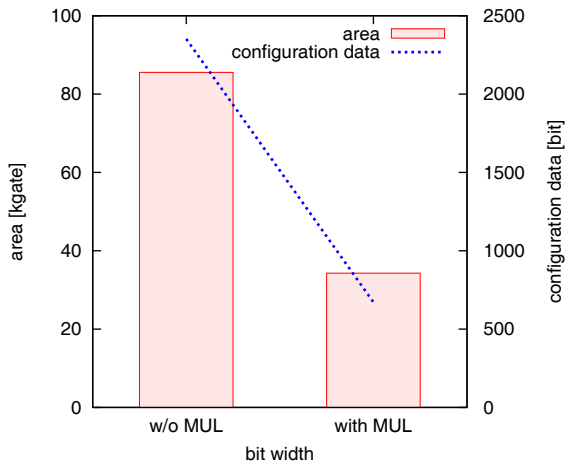

Fig. 11 Compilation results for DCT with different instruction sets

6. Conclusion

In this paper, a retargetable compiler which is helpful for exploration and evaluation of self-reconfigurable architectures is proposed. The proposed compiler provides efficient evaluation and comparison for various kinds of architectures on a common platform. The compiler analyzes an input C code and optimizes it with GCC, generates a DFG, converts to a netlist of ALUs, places and routes ALUs, and finally produces configuration data for a target architecture.

We also demonstrate some examples of architecture evaluation using the proposed compiler. This shows that our compiler enables quantitative evaluation, which is helpful for comparing various self-reconfigurable architectures. The demonstration also shows the compiler’s retargetability, *e.g.*, parameterized bit width and wire tracks and different instruction sets for ALUs.

Using the proposed compiler and the evaluation platform, we will make intensive experiments on various architectures and applications in order to clarify the architecture suitable for a certain application and/or the application executed efficiently on a certain architecture, and finally, we will develop a new coarse-grained self-reconfigurable device which is highly optimized for a practical application domain. We are also planning to make enhancements of our compiler, including parallelization and/or pipelining of loops, task scheduling for dynamic reconfiguration, and co-design of reconfigurable fabric and conventional processor based on application profiling.

Acknowledgments

This work is partly supported by the Japan Society for the Promotion of Science (JSPS) the 21st Century COE Program (Grant No. 14213201) and Grants-in-Aid for Scientific Research (B) 17300016 from JSPS. This work also supported by the VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Mentor Graphics, Inc.

References

[1] National Institute for Standards and Technology, “Announcing the Advanced Encryption Standard (AES)”, Federal Information Processing Standards Publication, vol.197, Nov 2001.

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

[3] W. H. Chen, C. H. Smith, and S. C. Fralick, “A Fast Computational Algorithm for the Discrete Cosign Transform”, The IEEE Transactions on Communications, vol.COM-25, no.9, pp.1004-1009, Sep 1977.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”, ACM Transactions on Programming Languages and Systems, vol.13, no.4, pp.451-490, Oct 1991.

[5] M. Y. Hsiao, “A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes”, IBM Journal of Research and Development, vol.14, no.4, pp.395-401, Jul 1970.

[6] H. Ito, R. Konishi, H. Nakada, H. Tsuboi, Y. Okuyama, and A. Nagoya, "Dynamically Reconfigurable Logic LSI – PCA-2", IEICE Transactions on Information and Systems, vol.E87-D, no.8, Aug 2004.
[7] H. Ito, K. Oguri, K. Nagami, R. Konishi, and T. Shiozawa, "The Plastic Cell Architecture for Dynamic Reconfigurable Computing", In Proceedings of 9th International Workshop on Rapid System Prototyping, pp.39-44, Jun 1998.
[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing", Science, vol.202, no.4598, pp.671-680, May 1983.
[9] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori, and K. Nagami, "PCA-1: A Fully Asynchronous Self-Reconfigurable LSI", In Proceedings of 7th International Symposium on Asynchronous Circuits and Systems (ASYNC 2001), pp.54-61, Mar 2001.
[10] S. Kouyama, T. Izumi, H. Ochi, and Y. Nakamura, "A simulation platform for designing cell-array-based self-reconfigurable architecture", IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol.E90-A, no.4, pp.784-791, Apr 2007.
[11] C. Y. Lee, "An Algorithm for Path Connection and its Applications", IRE Transaction on Electronic Computing, vol.EC-10, pp.346-365, Sep 1961.
[12] S. Morioka and A. Satoh, "An Optimized S-Box Circuit Architecture for Low Power AES Design", In Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, pp.172-186, Aug 2002.
[13] M. Motomura, "Dynamically Reconfigurable Processor Architecture", Microprocessor Forum, Oct 2002.
[14] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi, "Plastic Cell Architecture: A Scalable Device Architecture for General-Purpose Reconfigurable Computing", IEICE Transaction on Electronics, vol.E81-C, no.9, pp.1431-1437, Sep 1998.
[15] D. Novillo, "Tree SSA—A New Optimization Infrastructure for GCC", In Proceedings of the GCC Developer's Summit, pp.181-193, May 2003.
[16] D. Novillo, "Tree SSA—A New High-Level Optimization Framework for the GNU Compiler Collection", In Proceedings of the Nord/USENIX Users Conference, Feb 2003.
[17] T. Sugawara, K. Ide, and Tomohiro Sato, "Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology", IEICE Transactions on Information and Systems, vol.E87-D, no.8, pp.1997-2003, May 2004.
[18] H. Tsutsui, A. Tomita, S. Sugimoto, K. Sakai, T. Izumi, T. Onoye, and Y. Nakamura, "LUT-Array-Based PLD and Synthesis Approach Based on Sum of Generalized Complex Terms Expression", IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol.E84-A, no.11, pp.2681-2689, Nov 2001.

Masayuki Hiromoro received his B.E. degree in Electrical and Electronic Engineering from Kyoto University in 2006. Presently, he is a master course student at Department of Communications and Computer Engineering, Kyoto University. He is a student member of IEICE, IPSJ, and IEEE.

Shin'ichi Kouyama received his B.E. degree in Electrical and Electronic Engineering and M.E. degree in Communications and Computer Engineering from Kyoto University in 2003, and 2005, respectively. Presently, he is a doctor course student at Department of Communications and Computer Engineering, Kyoto University. He is a student member of IEICE and IEEE.

Hiroyuki Ochi received the B.E., M.E., and Ph.D. degrees in Engineering from Kyoto University in 1989, 1991, and 1994, respectively. In 1994, he joined Department of Computer Engineering, Hiroshima City University as an associate professor. Since 2004, he has been an associate professor of Department of Communications and Computer Engineering, Kyoto University. He is a member of IEICE, IPSJ, and IEEE.

Yukihiro Nakamura received his B.S., M.S. and Ph.D. degrees in Applied Mathematics and Physics from Kyoto University, in 1967, 1969 and 1995, respectively. From 1969 to 1996, he was with Electrical Communications Laboratories, NTT. In NTT he engaged in research and development of the behavioral description language "SFL" and the High-Level Synthesis System "PARTHENON". Concurrently, he was a guest professor at Graduate School of Information Systems, University of Electro-Communications. In 1996, he joined Graduate School of Informatics, Kyoto University as a professor. Since 2007, he has been a professor of Research Organization of Science and Engineering, Ritsumeikan University. He has also been a coordinator of Synthesis Corporation since 1998. He received Best Paper Award of IPSJ, Okochi Memorial Technology Prize, Minister's Prize of the Science and Technology Agency and Achievement Award of IEICE in 1990, 1992, 1994 and 2000, respectively. He is a fellow of IEEE and a member of IEICE, IPSJ and ACM.