Black-Diamond: a Retargetable Compiler using Graph with Configuration Bits for Dynamically Reconfigurable Architectures

Vasutan Tunbunheng Hideharu Amano
Dept. of Information and Computer Science,
Keio University
Yokohama, 223-8522, Japan
vasutan@am.ics.keio.ac.jp hunga@am.ics.keio.ac.jp

Abstract— For developing design envionment for various types of Dynamically Reconfigurable Processor Arrays (DRPAs), the GCI (Graph with Configuration Information) is proposed to represent configurable resource in the target dynamically reconfigurable architecture. The function unit, constant unit, register, and routing resource can be represented in the graph as well as the configuration information. The restriction in the hardware is added in the graph by using "DisCounT" port which is limited the possible configuration bits at the port controlled by the other ports.

A prototype compiler called Black-Diamond with GCI is now available for three different DRPAs. It translates data-flow graph from C-like front-end description, applies placement and routing by using the GCI, and generates configuration data for each element of the DRPA in the form of multicasting. Implementation results of simple applications show that Black-Diamond can generate reasonable designs for three different architectures.

I. Introduction

In recent years, coarse grained Dynamically Reconfigurable Processor Arrays (DRPAs) have been received an attention as a flexible and efficient off-loading engine for various types of System-on-Chips (SoCs). Some devices are commercially available [1, 2, 3, 4], and some of them have been integrated into digital appliances [5].

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 an PE array which enables time-multiplexed execution is introduced. Some of them use a multicontext facility which can change its configuration with a single clock cycle.

Unlike common FPGAs, in which the island-style structure using Look Up Tables (LUTs) with 4 or 5 inputs are commonly used, there are exist wide design choices in DRPAs, such as the PE granularity, the number of hardware contexts which can be switched dynamically, the total amount of wiring resource, and the size of PE array itself. Our performance evaluation results revealed that the optimal PE array size considering the area and power consumption is different for each application [6]. Thus, there is no all-around architecture in DRPAs, and the structure should be configurable or customizable for its main target application. Since DRPAs are embedded into an SoC, their architecture should be customized at the design time.

For such customized DRPAs, the programming environment, especially a compiler, must be also customized. A retargetable compiler which generates configuration data from C-like description is the most important component for such customized DRPAs. Unlike compilers for common CPUs, the core of compilers for DRPAs is mapping and place/routing functionality like common FPGA design tools. A direct graph which represents the target architecture is used. Unlike uniform FPGAs, DRPAs equip various types of components each of which has different restrictions. For example, the interconnections between networks and components are often limited by reducing configuration data. Even in the switching element, the flexibility is often limited. Although some retargetable compilers for DRPAs have been announced [7] [8], it is difficult to optimize the routing and generate the configuration data from such direct graph with various restrictions.

Here, we propose a retargetable compiler which can generate various kind of configuration data of DRPAs easily. In the compiler, a special graph called *Graph with Configuration Information (GCI)* whose node has configuration bits for generating the configuration data is used. By applying a technique to control selecting configuration at each node called *Disabling Configuration Testing (DisCounT)*, the restriction in the target architecture can be easily treated. The registers which can be used for transferring data between contexts can be also elegantly represented.

The organization in this paper is followed: the GCI is proposed and discussed in Section II. The coarse-grained multicontext reconfigurable architectures called *MuCCRA* as a case study is shown in Section III. The simple retargetable compiler called *Black-Diamond* was developed and used to map application into the architecture is described in Section IVand V. Finally, this work is concluded in Section VI.

II. Graph with Configuration Information

A. Graph Representation of Target Architectures

Deep into the long history of placement and routing, both fine-grained and coarse-grained architectures need a directed graph for representing routing resource of target architecture. There are nodes at input and output ports of Functional Unit (FU), and switching modules are also represented with a collection of nodes. Figure 1 shows a graph representing an array structure with four FUs connected in an island style interconnection

network. In this figure, a path between a source node to a sink node represented with bold lines is found in the router, and after the routing, the configuration bits to control transferring at each node is generated.

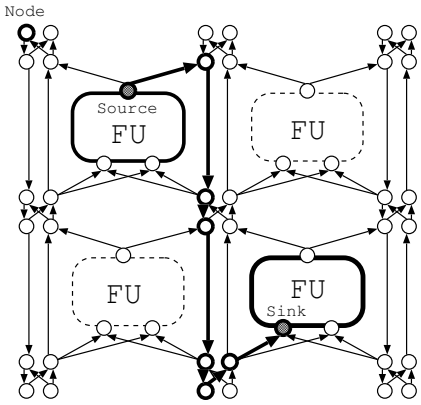


Fig. 1. Example of direct graph for representing routing resource (the bold line shows example of routing path from source port to sink port)

In such graphs, nodes without any configuration information are used for applying routing algorithms separately from configuration generation. This elegant approach adopted in DRESC [7][8] is suitable for an ideal architecture without any routing restriction. However, in most dynamically reconfigurable systems, there are various kind of restrictions for reducing configuration data and representing register files or distributed shared memory modules. They make both routing and configuration data generation difficult especially when the graph without any configuration information is used for representing the target architecture.

B. Graph with Configuration Information

Here, a directed graph called *Graph with Configuration Information (GCI)* is proposed for representing a target architecture. In the GCI, every node called *port* has input links which has a certain digit of data corresponding to its configuration. Links in the GCI are real links which are the target of the routing, or virtual links which are fixed and not the target of routing. The configuration data associated to the input links is defined so that they can be directly used for the operation code of Functional Units (FUs) or control code of multiplexers. In other words, all possible configuration bits of the target architecture are embedded in the GCI. Note that, the selecting input data is broadcasted to all output links.

Figure 2(a) shows the representation of an FU with four operations. The input port A and B receive data from other FUs or network. Each input link has configuration bits for selecting the input data, thus, it is used as a control information of input functions. Here, four operations of the FU are represented with 2 bits and associated with virtual links of the output port. Fixing the operation is corresponding to the selection of a virtual link. For a function which generating a constant data, it can be represented as Figure 2(b). Each virtual input link is associated with a constant data, and selecting one of them is deciding generated constant data. Note that, the configuration bits at the virtual links are corresponded to the constant bits.

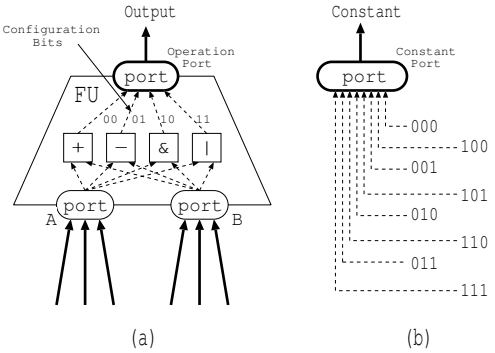


Fig. 2. (a) Operation port, and (b) Constant port (the dash arrows are virtual links and not actual connected in the graph because the input data does not transfer from any exist port)

From the results of placement and routing, the configuration data is directly generated by just combining the configuration bits of a selected link at each port in a specific order. If the target component is not used, the port generates *default configuration bits* to fulfill the entire code of the configuration data.

C. Disabling Configuration Testing

In order to embed the whole information of the target architecture into GCI, the restriction of the target architecture must be represented. For example, a register file is a common component of DRPA, and often consisting of two-port structure, that is, a port A can be used both for reading and writing, but port B can be used only for reading. In this case, when port A is used for writing data, it cannot be used for reading simultaneously except that the reading address and writing address are the same. In order to represent such a situation, a *Disabling Configuration Testing (DisCounT) port* and *control link* are introduced. Each control link sends information whose input links of the target DisCounT are disabled. This information is depending on the selected input link of the source port. A control link is activated when an input link of the source port is selected temporarily and the corresponding inputs of the DisCounT port are disabled. A DisCounT port can be controlled by multiple source ports. When multiple control links are activated, only control information which results at least one enable input at the DisCounT port is accepted. If the control information is not accepted, the selection of some source ports are canceled. Note that, the DisCounT port must select an enable input for configuration generating.

Figure 3 shows a GCI representation of a one-port register file with 4 entries. A register address is used for both reading and writing. The shaded port is the DisCounT port which generates the address instead of output port. If input from "R3" is selected at the "Output" port in Figure 3, the control link disables all virtual links other than R3 at the DisCounT port. In this case, only a control link from "Write Port" corresponding to R3 is acceptable. With similar manner, it is found from the graph that connections are allowed only when the register selected in the "Output" and the register address of "Write Port" are the same.

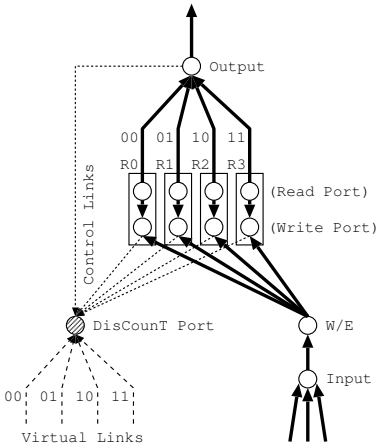


Fig. 3. An example of register file with four entries (the read port receives input data from the write port of previous context)

III. A Case Study: MuCCRA architecture

Here, as a case study, an example of representing a dynamically reconfigurable processor called MuCCRA (Multi-Core Configurable Reconfigurable Architecture) [9] based on the GCI is shown. Since MuCCRA is a project investigating an optimal structure of DRPA for a given application, several prototypes with different structures have been designed. Three different MuCCRA structures: MuCCRA-1, MuCCRA-2 and MuCCRA-D are treated as targets. Since MuCCRA-1 has been explained many articles [9], here, MuCCRA-2 which is 16 bits architecture with 16 processing elements implemented with ASPLA's 90nm process technology is introduced as an example.

A. PE Architecture

The basic building unit of MuCCRA-2 is a Processing Element (PE) shown in Figure 4(a) and Distributed Memory Module (MEM) shown in Figure 4(b). Each PE has a programmable PE-Core, connection blocks, and a context memory. In the PE-Core, like a lot of existing DRPA devices, a data manipulator called Shift & Mask Unit (SMU), an Arithmetic Logic Unit (ALU), and a Register File Unit (RFU) are provided. Note that, the ALU includes multiplier operation. MuCCRA-2 uses a 16 bits architecture, and links for data communication are basically 16 bitwidth. Each PE is connected with global routing wires via connection blocks. The connection blocks pick up the data in global routing wires, and distribute to all functional units of a PE-Core.

Each PE equips its context memory which provides multiple sets of configuration data corresponding to the operation of ALU, SMU, register file, and interconnection. The Central State Controller (CSC) broadcasts a context pointer to all PEs. The context is read from the context memory according to the context pointer, and they are reconfigured in parallel. This type of dynamic reconfiguration is called a multicontext scheme, and a lot of current devices support it. In the multicontext devices, the dynamic reconfiguration can be done in only one clock cycle by distributing the context memory into each reconfigurable module. In MuCCRA-2, 16 contexts can be held in the context memory.

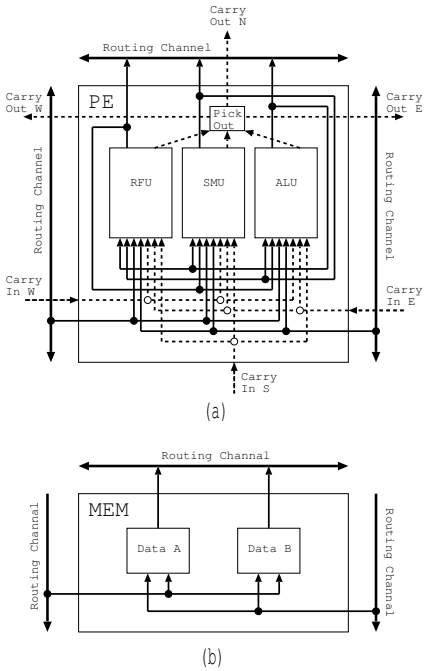


Fig. 4. A Target (a) PE-Core and (b) MEM and corresponding GCI

The GCI models of ALU and SMU are shown in Figure 5 and 6, respectively. In the ALU, some functions are selected depending on the carry signal since they can pass "Carry In" to output. So, a DisCounT port for selecting the operation is controlled by the link from the port "Carry Out".

In SMU graph, 4 DisCounT ports are used to represent two types of operation code. In order to avoid the redundancy in configuration code, SMU uses two modes of operations: "Short-Constant" with 6 bits operation code and 14 bits constant value, and "Long-Constant" with 4 bits operation code and 16 bits constant value. DisCounT ports are used to selecting one from two modes. In this case, the virtual input which generates an empty string (NULL) is enabled when the corresponding mode is not selected, and thus, all other virtual input links are disabled in DisCounT ports.

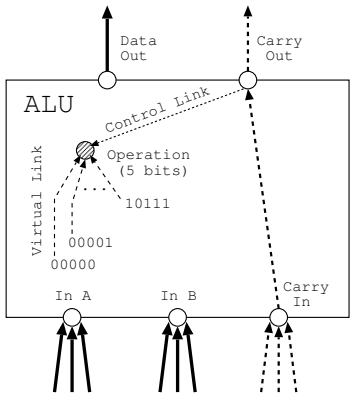


Fig. 5. A Target ALU architecture and corresponding GCI

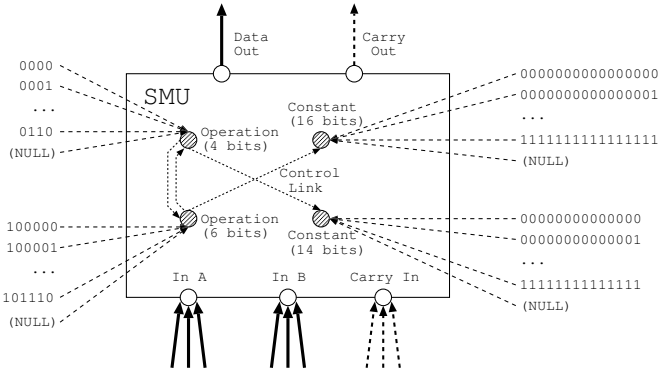
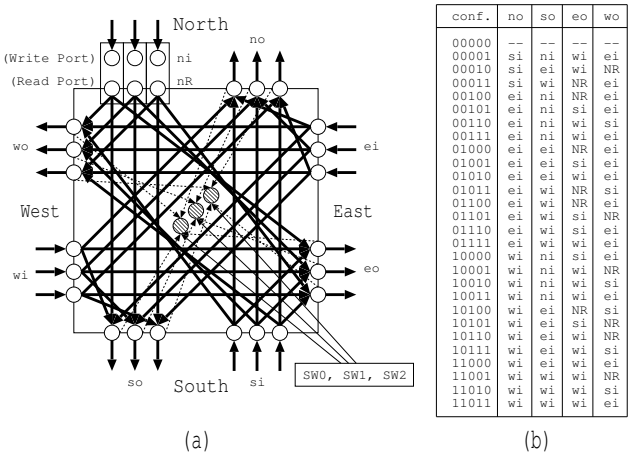


Fig. 6. A Target SMU Architecture and Corresponding GCI graphs



conf.	no	so	eo	wo
00000	--	--	--	--
00001	si	ni	wi	ei
00010	si	ei	wi	NR
00011	si	wi	NR	ei
00100	ei	ni	NR	ei
00101	ei	ni	si	ei
00110	ei	ni	wi	si
00111	ei	ni	wi	ei
01000	ei	ei	NR	ei
01001	ei	ei	si	ei
01010	ei	ei	wi	ei
01011	ei	wi	NR	si
01100	ei	wi	NR	ei
01101	ei	wi	si	NR
01110	ei	wi	si	ei
01111	ei	wi	wi	ei
10000	wi	ni	si	ei
10001	wi	ni	wi	NR
10010	wi	ni	wi	si
10011	wi	ni	wi	ei
10100	wi	ei	NR	si
10101	wi	ei	si	NR
10110	wi	ei	wi	NR
10111	wi	ei	wi	si
11000	wi	ei	wi	ei
11001	wi	wi	wi	NR
11010	wi	wi	wi	si
11011	wi	wi	wi	ei

Fig. 8. (a) Switch box architecture, and (b) Routing table

B. Array Architecture and Switching Elements

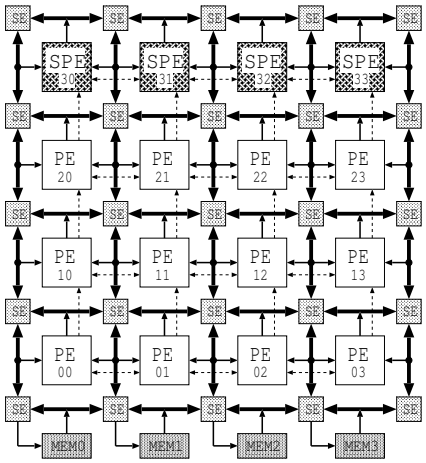


Fig. 7. MuCCRA-2 array structure

An island-style interconnection structure like traditional FPGAs is adopted in MuCCRA-2. As shown in Figure 7, an island-style 2-dimensional interconnection is provided, and each PE is surrounded by programmable routing wire segments. As mentioned, connection blocks mediate the connection between PEs and global routing resources. On the intersection of a vertical and a horizontal channel, a Switching Element (SE) is placed. The SE is a set of simple programmable switches in which an entering link is connected to the other SEs. In MuCCRA-2, three bi-directional routing channels are provided.

The SE can route data in 4 directions (NEWS) base on crossbar connection without loop back to the input direction. In MuCCRA-2, the fully routing capability is not allowed for reducing the configuration bits, and only the switching pattern shown in the table (Figure 8(b)) is allowed.

The *no,so,eo, and wo* are output ports and the *ni,si,ei, and wi* are input ports. There is a register at the north direction to store input data before transferring to east and west direction to avoid combinatorial loop in the interconnection network. The *ni* is write port to transfer input data to read port ("nR") of the next context. The configuration bits *00000* is used in the case that no input to be routed by sending zero value instead to save energy.

The corresponding GCI is shown in Figure 8(a). DisCounT ports ("SW0", "SW1", and "SW2") are used to represent the restriction shown in Figure 8(b). The connecting path can be established from input to output if there is still one or more enable inputs after other control information disables virtual input links at the DisCounT port.

C. Array Architecture of MuCCRA-D

In three MuCCRA architectures, the MuCCRA-1 is almost the same structure as the MuCCRA-2 except number of routing channal, bitwidth, and no restriction on the SE module. The multiply units do not include in ALU but connected on the left side of PE array.

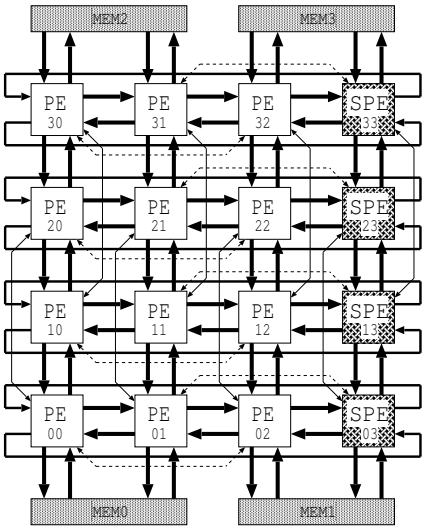


Fig. 9. MuCCRA-D array structure

Unlike island-style interconnection adopted in MuCCRA-1 and MuCCRA-2, MuCCRA-D architecture uses Nearest Neighbour (NN) interconnection network in order to transfer data be-

tween the PEs quickly as shown in Figure 9. There are 3 routing channels in 4 directions (NEWS) input to ALU, SMU and RFU at the neighboring PE respectively. The restriction in GCI is applied to limit transferring output of each component in the PE-Core to only 1 direction for reducing the configuration data size. In order to reduce the number of hops for transferring data to distant PEs, one routing channel is connected on both horizontal and vertical direction to the next neighboring PE, while torus connection is available on only horizontal direction.

Register to store the output data at each component before transferring to the other PEs allows MuCCRA-D executing at high clock frequency. The MEM modules are connected on upper and lower side of the PE array, and the computation flow can go both up and down direction to exchange data between the memory modules unlike MuCCRA-1 and MuCCRA-2.

IV. A Retargetable Compiler: Black-Diamond

A. The outline of the compiler

We have developed a simple retargetable compiler called Black-Diamond based on the GCI, and now it can generate configuration codes of three different models of DRPA; MuCCRA-1, MuCCRA-2 and MuCCRA-D.

The flow of compilation is shown in Figure 10. Although common placement and routing algorithms used for FPGAs can be applied on the GCI, we adopt the simplest method in order to develop the retargetable compiler as quick as possible by making the best use of the characteritics of the GCI.

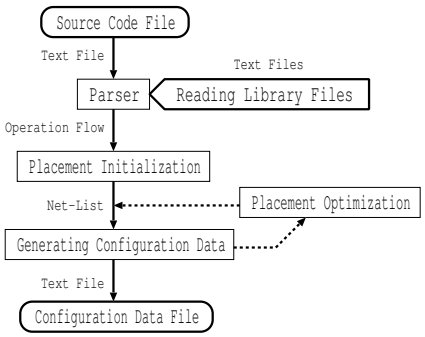


Fig. 10. Compile flow of the Black-Diamond Compiler

The target application is described in a C-like language as shown in Figure 11. After giving source code file into the compiler, library file corresponding to header file name in the source code is read out. It is consist of GCI and library functions to be placed in the target architecture. Once the application is mapped into the architecture, the compiler generates configuration data in text format as output.

Unlike common tools for FPGAs, it can fix the placement and routing at the same time. Since the GCI is a kind of constraints graph, searching the possible selection on the GCI is corresponding to fix everything. Like the common placement and routing algorithms, the Simulated Annealing (SA) algorithm [10] is used to find better selections iteratively. Once the optimized selection can be obtained, the configuration bits can be generated immediately from the GCI.

The configuration data can be generated in RoMultiC form [11] in order to reduce the number of configuration clock cycles. Since the MuCCRA architecture supports RoMultiC only on the PE and SE configuration, some ports in the element are declared to be bitmap type to indicate different configured position using for the RoMultiC form. The duplicated configuration data found in the different elements can be configured in parallel by combining the active bitmap patern. The port which does not select input link during the placement and routing can become the same selection as in another element to reduce the number of configurations.

B. The front-end Language

The parser used to translate the input source code to a data flow graph is developed by using LEX & YACC [12] in Linux. The data flow graph represents dependency by connecting the nodes corresponding to operations (library functions), and it is corresponding to the application. The output port of a node is linked to connect input port of the other operations by using a variable. The same variable name can be used to connect many different pairs of source and sink ports.

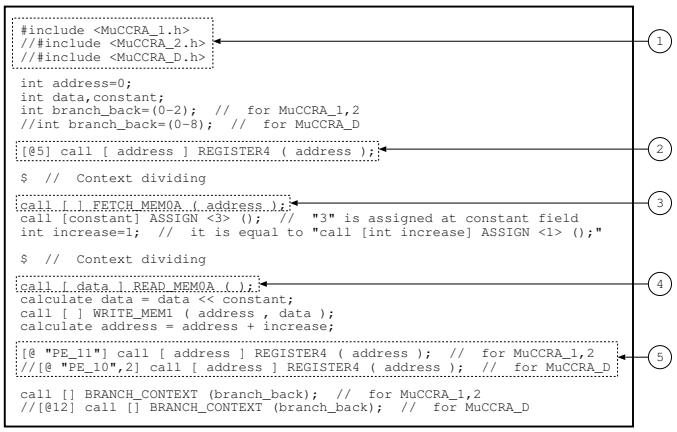


Fig. 11. An example input source code ((1) header file for indicating architecture name, (2 and 5) register used in looping, (3) giving reading address to memory, and (4) reading data from memory)

There are 3 parameter fields to call a function: *output field* (in front of function name), *input field* (behind function name), and *constant field* (between function name and input field if needed). The output field can return multiple variables since pointers and structures are not allowed, and the input field can receive multiple variables to link the return data from other functions. The constant field is provided to tell the compiler to select an input link at the constant port type in placement position corresponding to the giving constant value.

C. Placement Initialization

In order to map application into the architecture represented in GCI, the application is translated into a directed graph representing the data flow between computational nodes called *data-flow graph* as shown in Figure 12(a). In this research, a computational

node is called an *operation* since almost of library functions select an operation at the target placement (in ALU or SMU) to perform computation on the datapath.

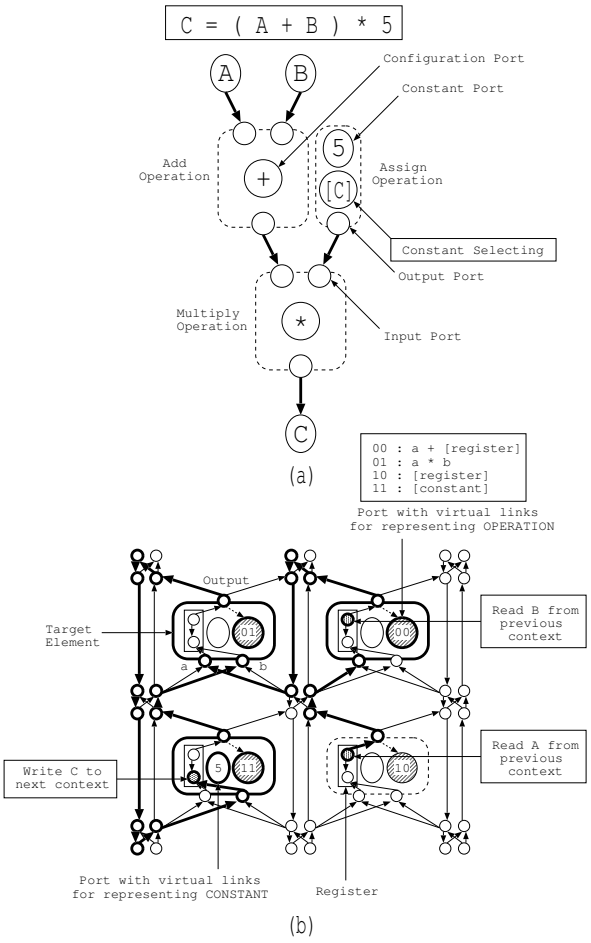


Fig. 12. (a) Operation flow, and (b) Example placing in the target architecture (The output of FU corresponding to the selecting at OPERATION port)

The operation has a list of ports in different types; *output ports*, *input ports*, *constant ports*, and *configuration ports* with selecting information. The output and input ports are source and sink ports to be connected to form the datapath in the data-flow graph. Some operations such as initial constant data (assign operation) or shifting operation require assigning constant value. The constant ports are used to indicate the port to be assigned the value. The configuration port comes with static constant value to configure the port corresponding to the computation.

In the placement and routing problem, there are many possible targets to place an operation in the target architecture. The possible target placement is called *target element* as shown in Figure 12(b). It is a group of ports to be refered as the port types in operation. Many operations can have the same target element. In this example, the target element are 4 FUs consisting of 7 ports and it can be refered to be placed all operations. The list of PEs become target elements and be attached to the operation used in order to find possible placement solution. At the output port of FU, there is input from the read port of register. With the same manner as in the ALU of MuCCRA architecture, when the output data is routed from the register, the control link disables all virtual input links except "10" at the "OPERATION" port. The selection at "OPERATION" port of lower right FU is set without placing operation.

From the data-flow graph, the connection transfer data from an output port of an operation to input port of the successor operation. The placement start searching from the first context. Each operation is tried to be placed in architecture according to the list of target elements in order to route the connection. If the first target element can not be placed, it tries to place in the next target element until finding the first possible placement that all inputs can be routed successful. A user can control to arrange the placement by inserting a pragma in source code to change the placement to the next possible target element instead. The port which is configured by placing the operation or routing is marked, and it can not be target of the next operation any more. In case that all target elements can not be placed, it tries to place in the next context. Then, the next operation returns to be placed from the first context again. The error is reported in the case that it requires exceeding limited available contexts in the target architecture, or in other words, the operation can not be placed.

D. Routing Algorithm

In order to find the minimum cost path in the GCI, the shortest-path algorithm with obstacle avoidance [13] is used. The path is searched by using Breadth First Search (BFS) to source port from the sink port. The algorithm is started by adding the sink port into the searching list which is empty when starting. If there is not any source port in the list, the minimum propagation cost port is replaced by its input ports which has not already searched yet. The process is repeated until the source port is found or no more un-reachable input port (not found). If there exists the connecting path, it can be obtained by backtracking from the finding the source port.

All ports on the connecting path are set to select configuration related to the backtracked input port. Since the GCI represents a constraint graph using DisCounT ports, input ports whose active control link to disable all the rest enable input links resulted from pre-routing connection at any DisCounT port are not added into the searching list. If all input ports on the connecting path become disable except the selected routing input port, the rest of enabling input port in pre-routing connection can be shared to route to the same source port.

E. The Example Application

Figure 11 shows an example application for shifting all data stored in MEM0 and writing result into MEM1 at the same address. By changing the header file declared at Figure 11(1), the same application can be mapped into different target architectures. The placement of each function can be automatically decided based on the restriction of GCI. The first possible placement position is selected, however, a user can control to place into the other positions by using pragma.

The "@" pragma controls shifting the placement to be another position. In the example at Figure 11(2), the shifting value is 5 and the followed calling function is placed into the sixth possible position. (placing at "PE11" in MuCCRA-1 and MuCCRA-2, and at "PE10" in MuCCRA-D) If the sixth possible position is not available in the first context, it tries to find in the next context

by routing input data via register automatically. The shifting pragma can also control the placement into a target element by indicating its name as shown at Figure 11(5).

The context looping can be performed by calling "BRANCH_CONTEXT" function to transfer negative value related to the next executed context. Even the register can be automatically assigned to transfer data to the next context, the current version of compiler can not automatically assign the same register for storing data between the last context in the loop and the first context in the next iteration. The "REGISTER4" function shown at Figure 11(2) and 11(5) is used to ensure that the register for holding the counting value "address" at the last context in the loop and reading at the first context in the loop are the same.

In many architectures, the PE array can read data from memory module with a clock delay. The reading address is sent, and the reading data can be available in the next context since the multicontext architecture can switch context within a clock. A user can insert pragma "$" between the function "FETCH_MEM0A" and "READ_MEM0A" to place them in different contexts.

Since the interconnection architecture of MuCCRA-1 and MuCCRA-2 are almost the same but different from the MuCCRA-D architecture, the number of context usage and placement are different. 3 contexts are used in the MuCCRA-1 and MuCCRA-2, while 9 contexts are used in the MuCCRA-D. In the case of MuCCRA-D, different set of pragmas which is commented out in the source code is used. By activing these lines, the code can be used for MuCCRA-D.

F. Graphic User Interface

Mapping of application on to the PE array can be controlled by the programmer in Black-Diamond compiler. As the initial placement, it maps operations to the first possible places to be routed automatically. However, the user may want to arrange the placement manually, for example, in order to reduce the loading configuration time in RoMultiC scheme or increasing usage of PE in a context. In this case, the user can insert pragma in the source code to control the place of mapping.

For such cases, Black-Diamond supports Graphic User Interface to show the placement and routing graphically. An example is shown in Figure 13. The target architecture is shown in 3-dimensional graphic. It can show the list of operations placed in each context with the same variable name in the source code. There is a pointer to show the port of operation in the target placed element corresponding to each variable at the parameter fields. By selecting input port, the routed path is highlighted in the picture, so, the user can easily trace the path. This GUI is also helpful for debugging.

V. Evaluation

It is difficult to demonstrate the benefit of the retargetable compiler, that is, how it can treat various target architectures easily. By using the GCI, Black-Diamond can treat three different types of DRPAs: MuCCRA-1, MuCCRA-2 and MuCCRA-D. Since the aim of MuCCRA project is investigating the design trade-off of various types of DRPAs, all three architectures are different as shown in Table I. Although an array with 16 PEs are used in all architectures, the bit-width, the type of operations, interconnection, and configuration bit structure are different.

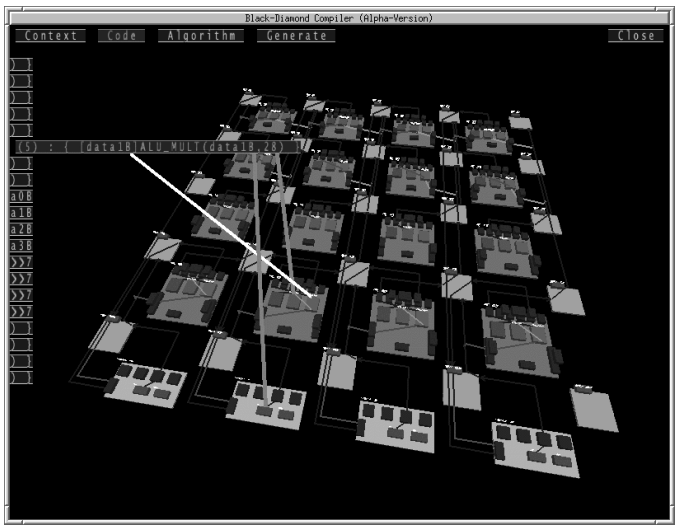


Fig. 13. Screen shot of the Black-Diamond Compiler

TABLE I
The difference between MuCCRA-1 and MuCCRA-2 ((a) Bit-width, (b) Contexts, (c) PE structure, (d) Interconnection, and (e) Process)

	MuCCRA-1	MuCCRA-2	MuCCRA-D
(a)	24 bits	16 bits	24 bits
(b)	64	16	64
(c)	Heterogeneous: including Multiplier PE	Homogeneous: All PE provides a Multiplier	Homogeneous: All PE provides a Multiplier
(d)	2 bi-direction	3 bi-direction	NN-interconnection
(e)	Rohm's 0.18um	ASPLA's 90nm	Rohm's 0.18um

Black-Diamond can generate configuration bits for all architectures just by changing the definition files. Here, application implementation examples on both architectures are shown. Since MuCCRA architecture is developed for multimedia processing, 3 applications are used here for evaluation: *Alpha-Blender* combines 2 input images depending on a constant *alpha*. Three pairs of color data (RGB) can be combined in parallel. *Discrete Cosine Transform (DCT)* is a part of JPEG coder, and treats 8×8 image matrices. First, 1 directional DCT is computed in the row direction, then the similar computation is done to the transposed matrix. Thus, it is consisting of two processes: 1D-DCT and transpose. *Contrast* is "Histogram Equalization" used to enhance contrast of input image. It includes 2 iterations, one is for uniforming the histogram and the other is for replacing color.

Table II, III, and IV show the required contexts, maximum clock frequency and execution time. Alpha-Blender is available in all architectures, DCT is not available on MuCCRA-2 since it

TABLE II
MAPPING AND EXECUTION RESULTS ON MUCCRA-1

Application	Contexts	Clk(MHz)	Exe. time (nsec)
Alpha-Blender	6	38MHz	6682
DCT:1D-DCT	12	27MHz	3240
DCT:Transpose	6	45MHz	924

TABLE III
MAPPING AND EXECUTION RESULTS ON MUCCRA-2

Application	Contexts	Clk(MHz)	Exe. time (nsec)
Alpha-Blender	5	90MHz	5643
Contrast	11	76MHz	5057

requires to be executed on 24 bits architecture, and instead of it, Contrast is implemented on MuCCRA-2. The clock frequency of MuCCRA-2 is larger than those of MuCCRA-1, since it is designed with more advanced process. In MuCCRA-D, there is register to store data before transferring it to other PEs, thus, all applications are executed in the same clock frequency.

The execution time of every application is superior to those from TI's DSP which works at 220MHz. Those results demonstrate that the practical applications can be developed using Black-Diamond with multiple architectures.

VI. CONCLUSIONS

The GCI is proposed to represent configurable resource in the target dynamically reconfigurable architecture. The function unit, constant unit, register, and routing resource can be represented in the graph as well as the configuration information. The restriction in the hardware is added in the graph by using "DisCounT" port which is limited the possible configuration bits at the port controlled by the other ports.

A prototype compiler called Black-Diamond with GCI is now available for three different dynamically reconfigurable architectures. It translates data-flow graph from C-like front-end description, applies placement and routing by using the GCI, and generates configuration data for each element of the DRPA in the form of multicasting. Implementation results of simple applications show that Black-Diamond can generate reasonable designs for three different architectures.

TABLE IV
MAPPING AND EXECUTION RESULTS ON MUCCRA-D

Application	Contexts	Clk(MHz)	Exe. time (nsec)
Alpha-Blender	11	125MHz	7200
DCT:1D-DCT	17	125MHz	928
DCT:Transpose	23	125MHz	184

REFERENCES

[1] M. Motomura, "A Dynamically Reconfigurable Processor Architecture," Microprocessor Forum, October 2002.

[2] T. Sugawara, K. Ide, and T. Sato, "Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology," IEICE Trans. on Inf. & Syst., Vol.E87-D", pp.1997-2003, 2004.

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

[4] Rapport, Inc., http://www.rapportincorporated.com

[5] Y. Kurose, et al., "A 90nm Embedded DRAM Single Chip LSI with a 3D Graphics, H.264 Codec Engine, and a Reconfigurable Processor," Hot Chips 16, September 2004.

[6] Y. Hasegawa, et al., "Performance and Power Analysis of Time-multiplexed Execution on Dynamically Reconfigurable Processor," Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS), April 2006.

[7] B. Mei, et al., "Dresc: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures," Proc. IEEE Intfl Conf. Field-Programmable Technology, IEEE Press, pp.166-173, 2002

[8] B. Mei, et al., "Architecture Exploration for a Reconfigurable Architecture Template," IEEE Design & Test of Computers pp.90-101, March.April 2005.

[9] Y. Hasegawa and H. Amano, "Design Methodology and Tradeoffs Analysis for Parameterized Dynamically Reconfigurable Processor Arrays," in Proc. of FPL.

[10] J. Rose, W. Klebsch, and J. Wolf, "Temperature Measurement and Equilibrium Dynamics of Simulated Annealing Placements," Computer-Aided Design, vol.9, no.3, pp.253-259, 1990.

[11] V. Tunbunheng, M. Suzuki, and H. Amano, "Data Multicasting Procedure for Increasing Configuration Speed of Coarse Grain Reconfigurable Devices," IEICE Transactions on Information and Systems, vol.E90-D, no.2, pp.473-481, February 2007.

[12] "YACC," http://dinosaur.compilertools.net

[13] H. Ishikawa, et al., "Shortest Path Algorithm on Parallel Reconfigurable Processor DAPDNA-2," Technical Report IEICE, NS2005-162, pp.17-20, March 2006.