Mixed-grained reconfigurable architecture supporting flexible reliability and C-based design

Hiroaki KONOURA^{1,6}, *Dawood ALNAJJAR*^{1,6}, *Yukio MITSUYAMA*^{2,6}, *Hiroyuki OCHI*^{3,6}, *Takashi IMAGAWA*^{4,6}, *Shinichi NODA*^{5,6}, *Kazutoshi WAKABAYASHI*^{5,6}, *Masanori HASHIMOTO*^{1,6}, *and Takao ONOYE*^{1,6}

^{1} Osaka University ^{2} Kochi University of Technology ^{3} Ritsumeikan University
^{4} Kyoto University ^{5} NEC Corp. ^{6} JST, CREST

***Abstract*—This paper proposes a mixed-grained reconfigurable architecture consisting of fine-grained and coarse-grained fabrics, each of which can be configured for different levels of reliability depending on the reliability requirement of target applications. Thanks to the fine-grained fabrics, the architecture can accommodate a state machine, which is indispensable for exploiting C-based behavioral synthesis to trade latency with resource usage through multi-step processing using dynamic reconfiguration. In implementing the architecture, the strategy of dynamic reconfiguration, the assignment of configuration storage and the number of implementable states are keys factors that determine the achievable trade-off between used silicon area and latency. We thus split the configuration bits into two classes; state-wise configuration bits and state-invariant configuration bits for minimizing area overhead of configuration bit storage. In addition, through a case study of FFT mapping, we experimentally explore the appropriate number of implementable states.**

***Keywords*—Coarse-grained reconfigurable architecture (CGRA), flexible reliability, behavioral synthesis, state machine**

I. Introduction

Coarse-grained reconfigurable architectures (CGRA) have been studied to fill the performance gap between FPGA and ASIC by reasonably limiting application domains and programmability. Recently, the reliability of reconfigurable devices is drawing attentions, since implementing mission-critical applications with high reliability on reconfigurable devices is highly demanded for saving NRE costs. Especially, soft errors are one of serious concerns threatening reliability of operating mission-critical applications. Soft errors include single event upset (SEU) where a charge occurring in memory elements causes a bit-flip and single event transient (SET) where a charge occurring in a combinational logic propagates to memory elements and causes a bit-flip. From reliability point of view, CGRA is inherently superior in soft error immunity to FPGA, since the amount of configuration bits is by orders of magnitude smaller than that of FPGA. [1–3] proposed several CGRAs with reliability consideration. Previously, we developed a reliability-configurable CGRA where the reliability level of each processing element (PE) can be chosen flexibly depending on applications and environments [4]. In [4], the trade-off between soft error immunity and area is successfully demonstrated on a 65nm test chip under irradiation.

Thus far, while many CGRAs have been proposed (e.g. [5–8]), their adoption for commercial use is limited, especially when compared to FPGAs despite of their larger power dissipation and area. CGRA is basically composed of an array of ALUs handling multi-bit operands, and is suitable for data-path implementation. On the other hand, it is not good at efficiently implementing one-bit operations that are often found in flag computation, conditional branching and state machine. Especially, the incompatibility with state machine implementation is a significant problem preventing CGRA from being widely used, since RTL designers and existing behavioral synthesis tools for ASIC and FPGA synthesize data-path circuits that are controlled by state machines. Consequently, CGRA is not taking full benefit of the IP reuse nor the standard ANSI C/C++ source codes available.

To overcome this issue, several CGRA architectures that are compatible with state machine implementation are proposed. For instance, [9] proposes DRP architecture which consists of a state transition controller and multiple PEs having 16 contexts and 8-bit/1-bit operators. This architecture enabled both one-bit operation and state machine implementation. Another example is XPP device [10], which has a configuration manager to enable the state transition in each tile. However, none of architectures have attained the compatibility with behavioral synthesis and reliability considerations. To expand the application domains of CGRA, an architecture having high compatibility with design automation tools and high flexibility in reliability to cover various applications is highly demanded.

In this paper, we propose a mixed-grain reconfigurable architecture that supports both behavioral synthesis and flexible reliability. The proposed architecture follows the concept of flexible reliability configuration presented in [4], which enables system designers to systematically trade off area for improving the soft error immunity without having a deep knowledge of reliability enhancement techniques. This work newly introduces one-bit PEs to implement a state machine that broadcasts the state signal to the array and dynamically reconfigures instructions given to CGRA. Consequently, designers can select one from various application implementations, e.g. a small area implementation with the large number of states or a low latency implementation with the small number of states.

We develop the architecture mentioned above which enables state-wise cycle-by-cycle dynamic reconfiguration, in contrast to multi-cycle reconfiguration [11]. To achieve this, we need to increase the capacity of local configuration memory in each PE in proportion to the number of states. In addition, to attain the immunity to soft errors, we need to introduce redundancy and error elimination mechanism into the configuration memory. For example, if triple modular redundancy (TMR) is adopted, the three-fold memory capacity becomes necessary. These two factors could tremendously increase the memory capacity for configuration bits, which could degrade area efficiency of the architecture.

To cope with this issue, we first adopt a strategy that the configuration bits for interconnection are not state-dependent, and only instructions to PE change according to the state signal. This strategy contributes to saving the memory capacity for configuration bits. Another important parameter to decide is the number of implementable states (#ImplSt). For exploring a wider trade-off between area and latency, the architecture that can implement the larger number of states is desirable. When #ImplSt is small, the obtainable trade-off between area and latency is limited. Meanwhile, when #ImplSt is excessively large, the area of configuration memory becomes significantly large. Especially, when a low latency implementation is selected from the trade-off, the number of used states is usually small. In this case, only a small portion of configuration memory is utilized, and consequently the unused memory results in area overhead. Furthermore, when #ImplSt is large, the state machine tends to be complex, and hence more one-bit PEs are necessary. This means that #ImplSt also affects the ratio of coarse- and fine-grained elements. Thus, #ImplSt must be carefully determined when implementing the architecture.

To investigate the appropriate #ImplSt, this work quantitatively evaluates two relationships between the number of states and resource usage and between #ImplSt and silicon area of a PE, respectively. Combining these evaluations, the achievable trade-offs between used silicon area and latency are illustrated for various #ImplSt, which suggests an appropriate #ImplSt.

The rest of this paper is organized as follows. Section II reviews related works, and Section III presents the proposed architecture. A quantitative evaluation on #ImplSt is shown in Section IV and a concluding remark is given in Section V. It should be noted that a preliminary version of the proposed architecture was implemented in 65nm process and the robustness of the test chip to irradiation is presented in [12].

II. Related Work

This section reviews conventional works of CGRA for attaining the compatibility with behavioral synthesis. None of conventional works have supported reliability enhancement for reliability demanding applications.

DAPDNA architecture [13] contains digital application processor (DAP) and distributed network architecture (DNA). DNA is composed of reconfigurable ALU, delay, and RAM elements having four configuration codes. DAP core triggers the state transition of DNA, and DAP will receive an interrupt signal from DNA notifying reconfiguration completion after several clock cycles. [14] developed DAPDNA-FWII, which compiles and maps a C source code into DAP and DNA.

XPP architecture [10] has one context in each PE and reconfigures it by configuration manager (CM). In this architecture, CM, which consists of a state machine and internal RAM for configuration caching, reconfigures processing array elements (PAE) within a few clock cycles through their individual configuration caches when triggered by event packets from PAEs. [15] introduced XPP-VC high-level compiler which maps C programs to XPP.

DRP architecture, which has multiple contexts in each PE and implementing state machines by context switch, and its behavior synthesis are presented in [9]. In this architecture, PEs containing 8-bit/1-bit operators and 16 different configuration codes, are reconfigured within one clock cycle by the broadcasted state number from a state transition controller (STC).

KAHRISMA architecture [16] is composed of coarse- and fine-grained encapsulated data-path elements (CG-EDPEs and FG-EDPEs) which are dynamically reconfigurable. CG-EDPE includes a context memory, a local sequencer, and a direct memory access. These components control the local state, i.e., dynamically reconfigure the operation of a processing block. [16] also presents a software framework which enables high-level compilation and mapping a C source code into CG- and FG-EDPEs.

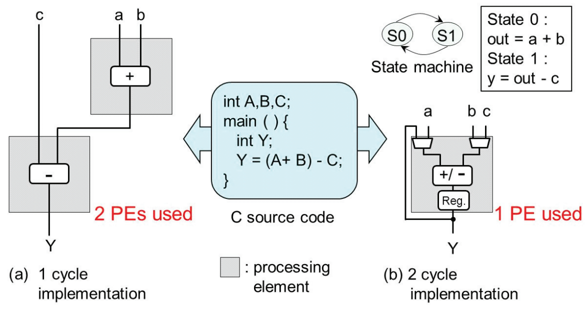

Fig. 1. Examples of implementations with different latency.

III. Proposed Architecture

This section firstly explains the compatibility with behavioral synthesis, and then describes the proposed architecture.

A. Compatibility with high level synthesis

Compatibility with behavioral synthesis requires architectural supports that help provide a trade-off between latency and resource usage (area). For this purpose, multi-step processing through dynamic reconfiguration should be utilized. The same blocks have to be used in different time slices to perform different operations. Fig. 1 shows a simple example demonstrating how multi-step processing is performed. The figure demonstrates how a C program can be implemented in one cycle and in two cycles. In the one cycle implementation of Fig. 1(a), two PEs are required (adder and subtractor). In this case, the configuration of the PEs is fixed, and the two PEs constantly perform the same operations until the device is reconfigured for another application. This is an ordinary output of common synthesis tools. Our architecture supports not only one-cycle implementation of Fig. 1(a) but also multi-cycle implementation. In the two-cycle implementation of Fig. 1(b), one PE including two multiplexers and a register, and a state machine are necessary. Dynamic reconfiguration of the PE is repeated. The state machine has two states, S0 and S1. During S0, the addition operation is selected to add a and b, and during S1, the c is subtracted from the output of the previous operation. Such a trade-off between latency and area obtained by various implementations allows attainment of various types of desirable specifications. In order to achieve this trade-off, the following elements are required: one-bit PEs to implement state machines, coarse-grained PEs to perform various types of data processing with dynamic reconfiguration depending on the state signal, register files to save temporal data, and large memories to store large bulk data. Here, although an embedded CPU might be able to control the state of coarse-grained PEs, we selected one-bit PEs for pursuing low-latency state control. With these elements, behavioral synthesis allows designers to explore the solution space and select an implementation that satisfies their requirements.

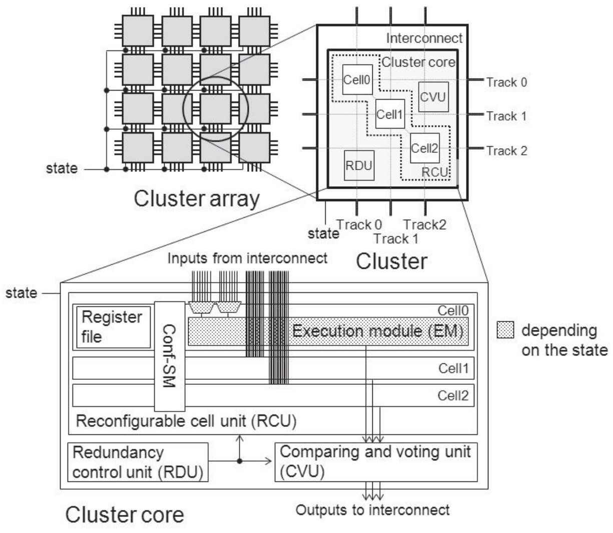

Fig. 2. Cluster and cluster interconnection.

B. Architecture design overview

The proposed architecture is composed of coarse-grained elements as ALU clusters, fine-grained elements as LUT clusters, and memory blocks called as MEM clusters. When an application is mapped the proposed architecture, data-paths are assigned to ALU clusters. Meanwhile, state machines and one-bit operations are assigned to LUT clusters. Each ALU cluster behaves as different functional units depending on the state. Due to this, on the other hand, ALU clusters themselves cannot hold register values which will be used after a while. To store the temporal intermediate data across the different states, register blocks called as REG clusters are also included in the proposed architecture.

In the architecture design, we selected the following strategy. ALU functionality and ALU operand selection are dynamically reconfigured according to the broadcasted state signals as shown in Fig. 2, which will be detailed in the next section, while the inter-cluster interconnection is unchanged. All the inter-cluster routings to provide data to clusters are fixed for all the states, and each ALU cluster selects a few data as operands depending on the state from all the data delivered to the ALU cluster. The reason why this strategy was selected is that the amount of configuration bits for inter-cluster interconnection is quite large, and it is difficult to multiply it by #ImplSt from area perspective.

The architecture has two global signals; context signal and state signal. Both of these global signals are generated by designated LUT clusters. The purpose of context signal is to switch the mapped application or algorithm, and then the inter-cluster interconnection is changed according to the context. This context signal is not discussed further in this paper.

Before explaining details of ALU, LUT, MEM and REG clusters, the treatment of the state signal in each cluster is briefly summarized. ALU cluster changes its functionality and data-path/flag operands according to the broadcasted state signal. Also, REG cluster selects input data and changes write address depending on the state signal, since the store of intermediate data depends on the state. On the other hand, the functionalities of LUT and MEM clusters are unchanged.

C. Details of four clusters

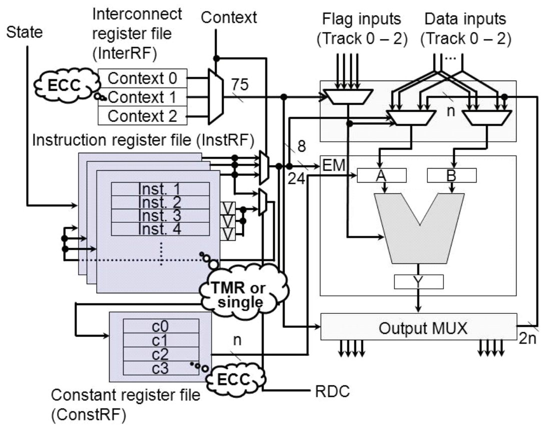
State
Interconnect register file (InterRF)
Context
Flag inputs (Track 0 – 2)
Data inputs (Track 0 – 2)
ECC
Context 0
Context 1
Context 2
75
n
Instruction register file (InstRF)
8
24
EM
A
B
Inst. 1
Inst. 2
Inst. 3
Inst. 4
V
V
V
TMR or single
Y
Output MUX
2n
c0
c1
c2
c3
n
ECC
RDC
Constant register file (ConstRF)

Fig. 3. Architecture of cell in ALU cluster.

1) ALU cluster: ALU cluster is derived from the reliability-flexible architecture proposed in [4]; however, it is highly improved to support cycle-by-cycle dynamic reconfiguration.

As shown in Fig. 2, an ALU cluster consists of a reconfigurable cell unit (RCU) processing various types of operations, a redundancy control unit (RDU) for flexible reliability, a comparing and voting unit (CVU), switches and wires. An RCU is composed of a configuration memory switching matrix (ConfSM) and three cells, each of which contains an execution module (EM), register files for storing configuration bits, and voters. In EM, arithmetic operation including multiplication, logic operation, and shift operation are performed.

The cluster interconnection has three tracks (Track 0 – 2), and each cell inside a cluster is placed on one of them. Thus, each cell in a cluster can be connected to the cells in adjacent clusters on the same track. The interconnection also has a diagonal track, connecting cells within one cluster. Switches to control these interconnections are implemented by multiplexers.

The cell architecture of ALU cluster is illustrated in Fig. 3. In order to implement dynamic reconfiguration with reduced area overhead, configuration bits are divided and stored in three types of register files: instruction register file (InstRF), interconnection register file (InterRF), and constants register file (ConstRF). The bit widths of InstRF, InterRF and ConstRF are 32, 75, and 16 bits, respectively. As mentioned earlier, instructions for ALU are locally stored in the cluster and the number of instructions stored is #ImplSt, where an instruction represents a set of ALU configuration bits for a single state. On the other hand, the configuration bits for inter-cluster connection stored in InterRF are fixed for all the states in each context. In this paper, InterRF is implemented so that it can store three contexts. ConstRF is used to store up to four constants that are required in the application, and one of four constants is selected by the 2-bit signal from InstRF. This implementation is selected for area reduction because at most only a few instructions need constants in most applications.

To attain soft error immunity, InterRF and ConstRF are protected with ECC. The selected code is single error correction/double error detection (SEC/DED) hamming code. For every read of InterRF and ConstRF, the error corrected bits are regularly restored in InterRF and ConstRF to prevent error accumulation. In addition, three contexts of InterRF and four constants of ConstRF are restored by re-writing the data itself

TABLE I. REDUNDANCY AND RELIABILITY TO SOFT ERRORS IN THREE OPERATION MODES IN ALU CLUSTER.

Operation mode	Redundancy		SEU in InstRF	SEU in EM	SET in EM	Utilization	
	InstRF	EM				#contexts	#cells
TMR	3	3	D & R	D & R	D & R	3	3
SMS	3	1	D & R	D	ND	1	1
SMM	1	1	ND	D	ND	3	1

D & R : Detection and recovery, D : Detection, ND : Does not detect

through an another SEC/DED encoder/decoder in rotation. On the other hand, InstRF is implemented with bit-wise TMR, since the path from the state signal to the register file output includes only a voter and its delay is small. This small delay is very important, since this delay is necessarily included in the critical path. This is the reason why ECC, which needs ECC decoder having large delay, was not selected for InstRF.

ALU cluster supports three operation modes: triple modular redundancy (TMR), single modular with single context (SMS), and single modular with multi-context (SMM), as summarized in Table I. These operation modes offer different capabilities of dynamic reconfigurability (no. of contexts) and throughput per cluster. TMR in which both InstRF and data-paths are triplicated provides the highest soft error immunity. Meanwhile, in SMS, only InstRF is triplicated but the data-path is singular. In both TMR and SMS, an SEU occurring in the InstRF will be repaired when the next configuration clock is given, since the voted value is re-written to the register file every configuration clock cycle. This configuration data is stored with bit-wise TMR, and therefore, multiple SEUs in different bits will also be corrected when the next configuration clock is given. On the other hand, in SMM, the voters are disabled, and three contexts are stored independently using three InstRFs, each of which is included in individual cells. With this implementation, users can flexibly choose the operation modes, depending on the importance of SEUs in InstRF and SEU/SET in EM.

In this architecture, as pointed out earlier, InstRF consisting of a larger number of words can accommodate a larger state machine, which enables area-efficient implementation trading larger latency. However, as #ImplSt becomes larger, the silicon area of ALU cluster increases, and the area overhead originating from the unused words of InstRF tends to be significant. This trade-off will be discussed in Section IV.

2) LUT cluster: An LUT cluster supports reliable and regular modes. The LUT cluster architecture is shown in Fig. 4. The LUT cluster contains three cells and each cell contains one configuration memory (ConfMem), LUT data registers, a pipeline register, wires and selectors. In reliable mode, the three ConfMems, three LUT data registers, and three data-paths in three cells are redundantly used. By this, SEUs in ConfMem and the LUT data registers are corrected by re-writing with the voters in the VC and VL. In regular mode, no redundancy is applied, and the given data are independently processed in each cell. Operation modes in LUT cluster are summarized in Table II. The operation modes of LUT cluster are controlled through a 1-bit TMRed value stored in the RDC.

LUT cell contains a 4-input LUT that can be cascaded with other cells to form a larger LUT. It can receive flags such as zero flag, overflow, underflow, most four least significant bits of the result, and carry generated in ALU cells, and can perform conditional operations, flag multiplexing, flag inversion, pipelining, and fixed outputs. Any single bit of n-bit ALU output can be provided to LUT cells via multi-cycle shifting. Thus, the architecture offers a significant amount of temporal flexibility in providing data to be able to take full

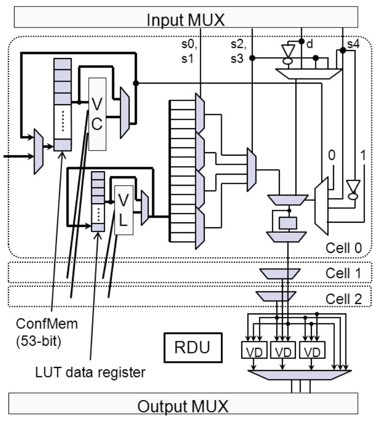

Fig. 4. LUT cluster architecture.

advantage of the fine-grained fabric. With receiving necessary 1-bit data, LUT clusters can perform one-bit operations and can form a state machine efficiently.

In the array, a set of LUT clusters, whose number depends on #ImplSt, are designated to output and broadcast the state signal. Similarly, another set of LUT clusters are responsible to output the context signal. Besides, LUT clusters are supposed to be organized in a two dimensional array forming LUT blocks, which makes the area of LUT block comparable to those of other clusters. LUT blocks, ALU, register and MEM clusters are placed in a two-dimensional array, whereas the interconnection among each cluster is not detailed in this paper.

3) MEM cluster: MEM cluster is composed of one 1,024-word $\times(n + k)$-bit two-port SRAM, where n represents the data-path width and k is the number of redundant bits. Although the SRAM itself is protected using SEC/DED hamming codes, the words which have not had a write access for a while are likely to accumulate multiple SEUs within a word, which results in uncorrectable errors. To avoid it, MEM cluster offers reliable mode in addition to regular mode. In regular mode, two ports of SRAM are independently utilized to read/write data. Meanwhile, in reliable mode, one port is used to read/write data, and the other port is designated for periodic overwriting through ECC decoder and encoder. Note that regular mode could be robust enough for applications that keep data for a short time, since SEU accumulation less likely happens.

MEM cluster has only one cell due to the size of the SRAM. However, it is compatible with the three-cell implementation of ALU cluster, the three-track flag and data interconnection, and the reliability modes of ALU cluster. MEM cluster also contains three tracks, all connected to the same cell. Input signals of SRAM such as address and enable are drawn with three tracks and voted in front of the SRAM macro. The read data of SRAM is also distributed to the three tracks. Herewith, single error points except for the inside of SRAM macro are minimized.

4) REG cluster: Architecture of REG cluster is shown in Fig. 5. REG cluster has three cells composed of a ConfMem, a w-word register file, wires and switches.

TABLE II. REDUNDANCY AND RELIABILITY TO SOFT ERRORS IN TWO OPERATION MODES IN LUT AND REG CLUSTERS.

Operation mode	Redundancy		SEU in ConfMem	SEU in LUT data register & register file	SET in data-path	Utilization	
	ConfMem	data-path				#contexts	#cells
Reliable	3	3	D & R	D & R	D & R	1	3
Regular	1	1	ND	ND	ND	1	1

D & R : Detection and recovery, ND : Does not detect

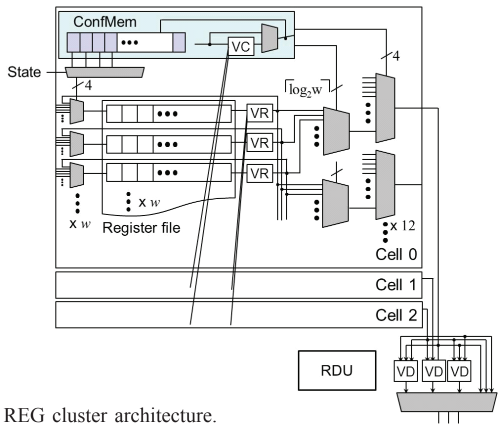

Fig. 5. REG cluster architecture.

Similarly to LUT cluster, REG cluster supports reliable and regular modes. In reliable mode, three ConfMems, three register files, and three data-paths in three cells are redundant. By this, SEUs in ConfMem and register files are corrected by re-writing through the voters in the VC and VR. Also, SETs in data-path can be corrected by the voters in the VD. In regular mode, no redundancy is applied, and the given data are independently processed in each cell. The reliability modes in REG cluster are as same as those in LUT clusters and summarized in Table II. The REG cluster operation mode is controlled through a 1-bit TMRed value stored in the RDC.

The REG cluster is responsible for storing and exchanging temporal data across different states. For this purpose, input data must be delivered to one of the registers depending on the state. This data delivery is achieved by w input multiplexers, and controlled by ConfMem, where ConfMem can store #ImplSt configuration sets. The relationship between #ImplSt and the area of REG cluster is evaluated in Section IV.

IV. EVALUATION RESULTS

As explained earlier, when implementing the proposed architecture, #ImplSt is a key parameter that determines the silicon overhead and the achievable trade-off between latency and used silicon area. This section experimentally evaluates the appropriate #ImplSt through a case study. Besides, the advantage of the flexible reliability is intensively evaluated in [4] and hence it is not evaluated in this paper.

In this evaluation, a behavioral synthesis tool [17] slightly customized for the proposed architecture is used to obtain various implementations with different latencies and ALU usages from a C source code of 512-point radix-8 FFT. Note that the bit width of ALU is supposed to be 16, and the operation frequency is set to 100MHz. Each cluster is implemented with Verilog HDL and synthesized with a 65nm cell library. The area of each cluster is estimated from the reports of logic synthesis tool. Reliability modes of ALU/LUT/REG clusters are configured with TMR/reliable/reliable, respectively.

A. Number of used clusters vs. number of states

The numbers of used ALU/LUT/REG clusters change depending on the number of states. Fig. 6 shows the relationship between the number of states and the number of ALU clusters in use. We can see that the number of ALU clusters in use decreases, which is a well-known relationship obtained by behavioral synthesis. Second, Fig. 7 shows the relationship between the number of states and the number of LUT clusters in use. As the number of states increases, the state machine becomes complex, and the number of LUT clusters increases.

TABLE III. REGISTERS FOR INTER-STATE DATA EXCHANGE.

#states	#registers for buffering	#REG clusters		
		$w = 4$	$w = 6$	$w = 16$
13	14	4	2	1
16	17	5	3	2
19	23	6	3	2
25	24	6	3	2
39	28	7	4	2

Moreover, Table III lists the required number of 16-bit registers and w-word REG clusters for inter-state data exchange. As the number of states increases, the number of registers for inter-state data exchange increases because the data exchange across the states happens more often. When the number of states is 39, the required number of four-word REG clusters is seven, on the other hand, that of 16-word REG clusters is two. The area of REG cluster, which depends on the number of words and #ImplSt, is evaluated in Section IV-C.

B. ALU cluster area vs. number of implementable states

An increase in #ImplSt involves a significant increase in the area of ALU cluster having TMRed InstRFs. Remind that #ImplSt is equal to the number of words of InstRF in ALU clusters. Fig. 8 shows the relationship between #ImplSt and the area of ALU cluster. As the number of states increases, the area of InstRFs linearly increases and gradually becomes dominant. When #ImplSt is 28, the area of ALU cluster reaches twofold compared with that of ALU cluster whose #ImplSt is 1.

C. Area of REG cluster

As shown in Fig 5, an increase in #ImplSt s enlarges configuration memories. Also, the number of words w is directly related to the sizes of register file and configuration memory. Fig. 9 shows the relationship between #ImplSt and the area of REG cluster for various numbers of word. There is a mostly linear relation between #ImplSt and the area of REG cluster. In a case that 39 states are implementable, the area of 16-word REG cluster (= 0.137 mm^2) is 3.7 times as large as that of a four-word REG cluster (= 0.038 mm^2), i.e., there is the proportional relation between the number of words and the area of REG cluster. Here, remind that the required number of REG clusters for application mapping is inversely proportional to the number of words in REG cluster in Table III. These results show the total area of REG clusters is not sensitive to the number of words, and we can select a reasonable number of words according to the capacity of inter-cluster interconnection.

D. Appropriate number of implementable states

We then evaluate the achievable trade-offs between used silicon area and latency for various #ImplSt. Fig. 10 shows

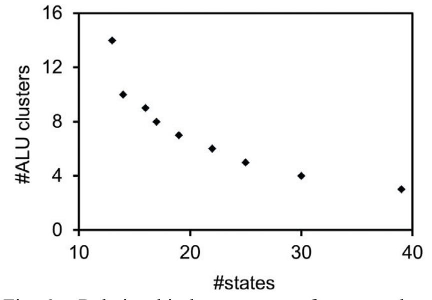

Fig. 6. Relationship between no. of states and no. of ALU clusters.

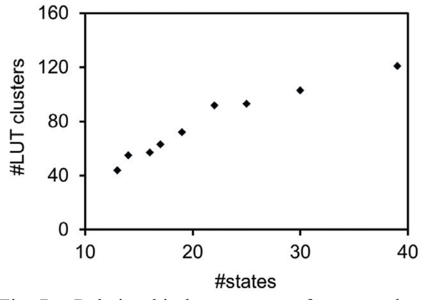

Fig. 7. Relationship between no. of states and no. of LUT clusters.

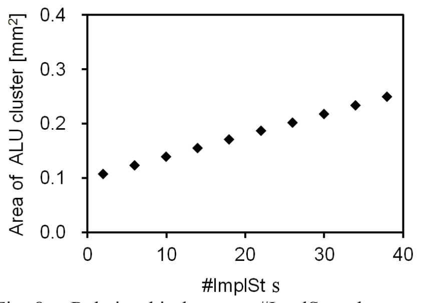

Fig. 8. Relationship between #ImplSt and area of ALU cluster.

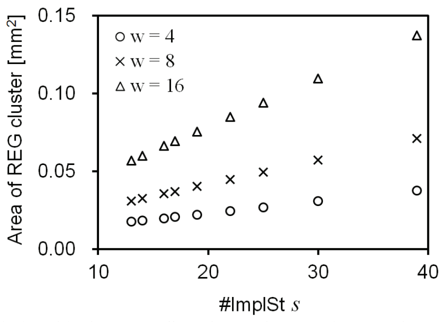

Fig. 9. Relationship between #ImplSt and area of REG cluster.

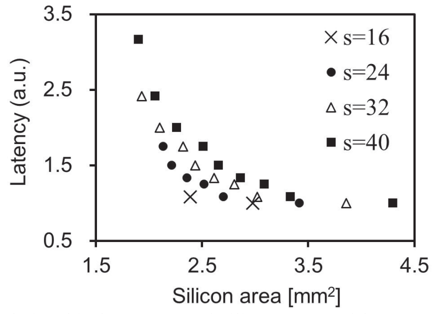

Fig. 10. Relationship between used silicon area and latency.

the results. The used silicon area includes those of used ALU, LUT, MEM, and REG clusters, and the dependency of ALU cluster area on #ImplSt (Fig. 8) is taken into consideration. Here, four MEM clusters, one of which is 0.111mm^2, are included. We assumed to use 16-word REG clusters. As can be seen from Fig. 10, when #ImplSt is small such as 16, a small-area implementation can be achieved; however, the achievable trade-off between used silicon area and latency is quite limited. On the other hand, when #ImplSt is large such as 40, a wide range of trade-off between used silicon area and latency is obtainable, whereas the used silicon area of the implementation pursuing the minimum latency becomes large. While the best #ImplSt depends on the requirements of area, performance and design flexibility, in this test case, the range from 24 to 32 is a reasonable number to take advantage of behavioral synthesis with limited overhead. Thus, the proposed mixed-grained architecture can obtain various implementations on the same cluster array making use of behavioral synthesis from a C source code.

V. Conclusion

We proposed a mixed-grained reconfigurable architecture supporting C-based behavioral synthesis and flexible reliability. We experimentally evaluated trade-offs between used silicon area and latency with various numbers of implementable states using 512-point radix-8 FFT as an application example. In this evaluation, the proposed architecture whose number of implementable states in the range from 24 to 32 can accommodate various implementations in latency and area obtained by behavioral synthesis.

References

[1] S. M. A. H. Jafri, et al., "Design of a fault-tolerant coarse-grained reconfigurable architecture: a case study" in *Proc. ISQED*, pp. 845 – 852, Mar. 2010.

[2] M. M. Azeem, et al., "Error recovery technique for coarse-grained reconfigurable architectures," in *Proc. DDECS*, pp. 441 – 446, Apr. 2011.

[3] T. Schweizer, et al., "Low-cost TMR for fault-tolerance on coarse-grained reconfigurable architectures," in *Proc. ReConFig*, pp. 135 – 140, Nov. – Dec. 2011.

[4] D. Alnajjar, et al., "Implementing flexible reliability in a coarse-grained reconfigurable architecture," *IEEE Trans. VLSI Systems*, in press. (http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6387625)

[5] S. C. Goldstein, et al., "PipeRench: a reconfigurable architecture and compiler," *IEEE Trans. Computers*, vol. 33, no. 4, pp. 70 – 77, Apr. 2000.

[6] M. Myjak, et al., "A two-level reconfigurable architecture for digital signal processing," *ScienceDirect Trans. Microelectronic Engineering*, vol. 84, no. 2, pp. 244 – 252, Feb. 2007.

[7] C. Ebeling, et al., "RaPiD - reconfigurable pipelined data-path," in *Proc. FPL*, pp. 126 – 135, Sept. 1996.

[8] Y. Mitsuyama, et al., "Area-efficient reconfigurable architecture for media processing," *IEICE Trans. Fundamentals*, Vol. E91-A, No. 12, pp. 3651 – 3662, Dec. 2008.

[9] T. Toi, et al., "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *Proc. ICCAD*, pp. 702 – 708, Nov. 2006.

[10] V. Baumgarte, et al., "PACT XPP - A self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167 – 184, Sept. 2003.

[11] L. Bauer, et al., "RISPP: rotating instruction set processing platform," in *Proc. DAC*, pp. 791 – 796, June 2007.

[12] D. Alnajjar, et al., "Reliability-configurable mixed-grained reconfigurable array supporting C-to-array mapping and its radiation testing," in *Proc. A-SSCC*, Nov. 2013 (to appear).

[13] T. Sugawara, et al., "Dynamically reconfigurable processor implementation with IPFlex's DAPDNA technology," *IEICE Trans. Inf. & Syst.*, vol. E87-D, no. 8, pp. 1997 – 2003, Aug. 2004.

[14] T. Sato, et al., "Implementation of dynamically reconfigurable processor DAPDNA-2," in *Proc. VLSI-TSA-DAT*, pp. 323 – 324, Apr. 2005.

[15] J. M. P. Cardoso, et al., "From C programs to the configure-execute model," in *Proc. DATE*, pp. 576 – 581, Mar. 2003.

[16] R. Koenig, et al., "KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Proc. DATE*, pp. 819 – 824, Mar. 2010.

[17] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, "Cyber"," in *Proc. DATE*, pp. 390 – 393, Mar. 1999.