JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.11, NO.3, SEPTEMBER, 2011 http://dx.doi.org/10.5573/JSTS.2011.11.3.207

Reconfigurable Multi-Array Architecture for Low-Power and High-Speed Embedded Systems

Yoonjin Kim

***Abstract*—Coarse-grained reconfigurable architecture (CGRA) based embedded systems aims to achieve high system performance with sufficient flexibility to map a variety of applications. However, the CGRA has been considered as prohibitive one due to its significant area/power overhead and performance bottleneck. In this work, I propose reconfigurable multi-array architecture to reduce power/area and enhance performance in configurable embedded systems. The CGRA-based embedded systems that consist of hierarchical configurable computing arrays with varying size and communication speed were examined for multimedia and other applications. Experimental results show that the proposed approach reduces on-chip area by 22%, execution time by up to 72% and reduces power consumption by up to 55% when compared with the conventional CGRA-based architectures.**

***Index Terms*—Embedded systems, Coarse-Grained Reconfigurable Architecture (CGRA), computing hierarchy, low power, high performance**

I. INTRODUCTION

As the market pressure of embedded systems compels the designer to meet tighter constraints on cost, performance, and power, the application specific optimization of a system becomes inevitable. On the other hand, the flexibility of a system is also important to accommodate rapidly changing consumer needs. To accommodate these incompatible demands while efficiently supporting the complex embedded-applications, domain-specific design has emerged as a suitable solution for embedded systems. Coarse-grained reconfigurable architecture (CGRA)-based embedded system is the very domain-specific design in that it can boost the performance by adopting specific hardware engines while it can be reconfigured to adapt to ever-changing characteristics of the applications.

In spite of the above advantages, the use of CGRA-based design has been prohibitive due to its significant area/power consumption. The area and power overheads are caused by large memory components and the computing block of many processing elements. There is also a performance bottleneck due to the conventional communication structure between processor and reconfigurable computing block that cannot adaptively support various applications. Therefore, reducing area/power and improving performance of CGRA-based system has been a serious concern.

In this paper, I propose a new reconfigurable computing hierarchy to design cost-effective CGRA-based embedded systems. The computing hierarchy consists of two reconfigurable computing blocks with two types of communication structure together. Based on the hierarchy, efficient communication structure between processor and reconfigurable computing blocks can reduce performance bottleneck in the CGRA-based architecture. In addition, the proposed reconfigurable array splits the computational resources into two groups (primitive resources and critical resources). Critical resources can be area-critical and/or delay-critical.

Manuscript received Jun. 10, 2011; revised July 22, 2011.
Dept. of Computer Science, Sookmyung Women's University, Korea
E-mail : ykim@sookmyung.ac.kr

Primitive resources are replicated for each processing element of the reconfigurable array, whereas area-critical resources are shared among multiple basic PEs in order to reduce more area of CGRA. Delay-critical resources can be pipelined to curtail the overall critical path so as to increase the system clock frequency. The two computing blocks have shared pipelined-critical resources. Such a sharing/pipelining structure provides efficient communication interface between them with reducing overall area.

This paper is organized as follows. After the related work in Section II, I describe coarse-grained reconfigurable architecture and loop pipelining as preliminary in Section III. In Section IV, I present the motivation of my approach. Then I propose the new reconfigurable computing hierarchy for CGRA in section V and the experimental results are given in Section VI. Finally I conclude the paper in the Section VII.

II. Related Works

In [1], Hartenstein summarized many CGRAs that had been suggested until 2001. Since then, many more new CGRAs have been continuously proposed and evolved. Most of them comprise of a fixed set of specialized processing elements (PEs) and interconnection fabrics between them. The run-time control of the operation of each PE and the interconnection provides the reconfigurability. However, such fixed architecture has limitations in optimizing the cost and performance for various applications. For example, Morphosys [2] consists of 8 × 8 array of Reconfigurable Cell coupled with Tiny_RISC processor through system bus. It shows good performance for regular code segments in computation intensive domains but requires large amount of area and power consumption. XPP configurable system-on-chip architecture [3] is another example. XPP has 4 × 4 or 8 × 8 reconfigurable array and LEON processor with AMBA bus architecture. A processing element of XPP is composed of an ALU and some registers. Since the processing elements do not include heavy resources, the total area cost is not high but the range of applicable domains is restricted. In addition, XPP shows significant communication overhead between the processor and RAA through the system bus. REMARC [5] is reconfigurable Multimedia Array Coprocessor that consists of a global control unit and an 8 × 8 array of nano processors. The nano processors do not also include heavy resources like XPP but it also restricts the range of applicable domains. However, the communication with main processor is faster than [2] or [3] because the processor can access the register-set by coprocessor data transfer instructions. However, limited size of the register-set causes heavy registers-array traffic restricting performance enhancement. ADRES [4] tightly couples a VLIW processor and a reconfigurable matrix through shared register file. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way, whereas the VLIW processor executes the non-kernel code by exploiting instruction-level parallelism. Even though it also provides the fast communication speed between VLIW and the matrix but the entire structure is very dependent on VLIW processor architecture and it require huge register file for the communication. Therefore, the performance is limited by size of the register file. In [12], authors have also used reconfigurable computing cache that is different from my proposed RCC. Their RCC block is LUT-based and used for both memory and computing units.

III. Preliminary

In this section, I briefly introduce coarse-grained reconfigurable architecture and loop pipelining that is a very representative computation model for CGRA. The loop pipelining is an essential concept to explain resource sharing and pipelining in CGRA described in section V.

1. Coarse-Grained Reconfigurable Architecture

Typically, a CGRA-based system consists of a main processor, a Reconfigurable Array Architecture (RAA), and their interface as Fig. 1. The RAA has identical processing elements (PEs) containing functional units and a few storage units. The PEs in the array are connected to the nearest neighbor PEs In addition, they have limited interconnections between non-neighboring PEs in order to perform column-wise or row-wise data-transfer efficiently. These interconnections are reconfigurable because input multiplexers in each PE have inputs from other PEs. The data buffer provides

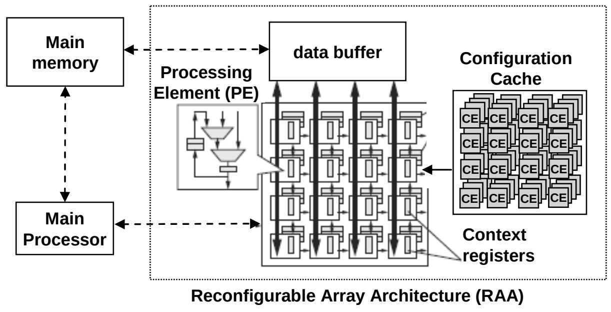


Fig. 1. Block diagram of general CGRA.

operand data to PE array through a high-bandwidth data bus. The configuration cache (or context memory) is composed of cache elements (CEs) and each CE provides context word to configure each PE.

2. Loop Pipelining

To represent the characteristics of loop pipelining [16], I examine the difference between SIMD and MIMD in the RAA with a simple example. I assume a mesh-based 4 x 4 coarse-grained reconfigurable array of PEs, where a PE is a basic reconfigurable element composed of an ALU, an array multiplier, etc. and the configuration is controlled by the words stored in the CE as shown in Fig. 2(a). In addition, I assume that Frame Buffer has simply one set having three banks and two read-ports and one write-port, supporting any combination of one-to-one mapping between the three banks and the three buses. Fig. 2(b) shows such a Frame Buffer and data bus structure, where the PEs in each row of the array share two read buses and one write bus. The 4 × 4 array has nearest neighbor interconnections as shown in Fig. 2(c) and each row or each column has a global bus as shown in Fig. 2(d).

Consider a square matrix *X* and *Y*, both of order *N*, and the computation of *Z*, an *N* element vector, given by

$$Z(i) = K \times \sum_{j=0}^{N-1} \{(X(i,j) + Y(i,j) \times C(j)\} \qquad (1)$$

where *i*, *j* = 0,1,…,*N*-1, *C*(*j*) is a constant vector, and *K* is a constant.

Consider *N*=4 for the mapping of the computation

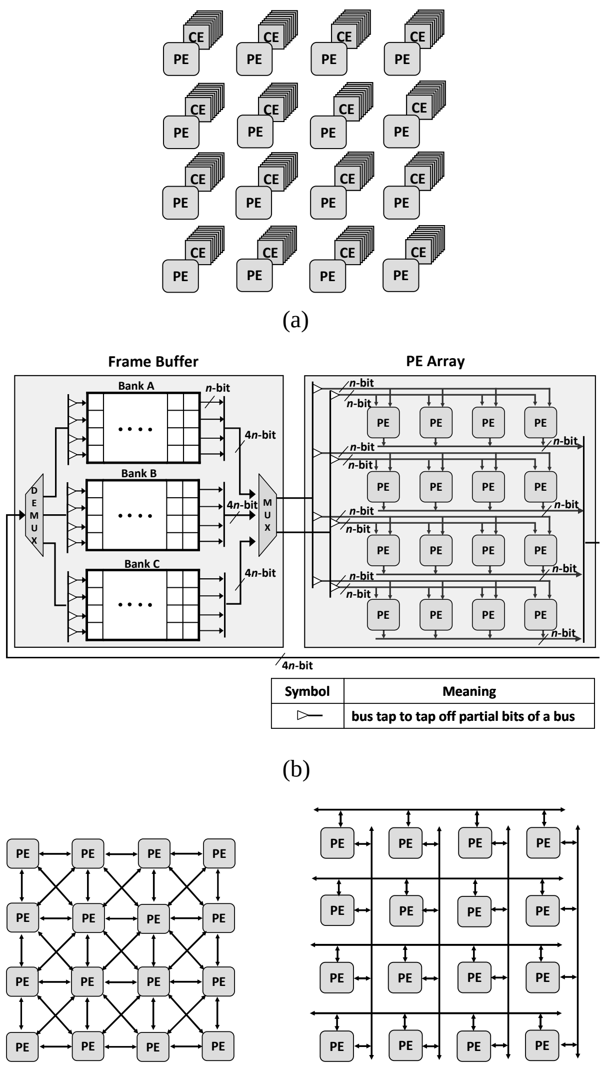


Fig. 2. 4 x 4 reconfigurable array. (a) Distributed cache structure, (b) Frame buffer and data bus, (c) Structure nearest neighbor interconnection, (d) Global bus interconnection.

defined in Eq. (1) on the 4 x 4 PE array and let the computation be given as a C-program (Fig. 3(a)). It is assumed that the input matrix *X*, *Y*, constant vector *C* and output vector *Z* are stored in the arrays *x*[*i*][*j*], *y*[*i*][*j*], *c*[*j*] and *z*[*i*], and *z*[*i*] is initialized to zero. Fig. 3(b) shows parallelized code for execution on the array as shown in Fig. 4, where I assume that matrix *X* and *Y* have been loaded into the Frame Buffer (FB) and all of the constants (*C* and *K*) have been already saved in a register file of each PE. Vector *Z* is stored in the FB after it has been processed by the PE array as shown in Fig. 4(a).

```
for (i = 0; i <= 3; i = i+1)
{
      for (j = 0; j <= 3; j = j+1)
            z[i] = (x[i][j]+y[i][j])*c[j] + z[i];
      z[i] = K* z[i];
}
```

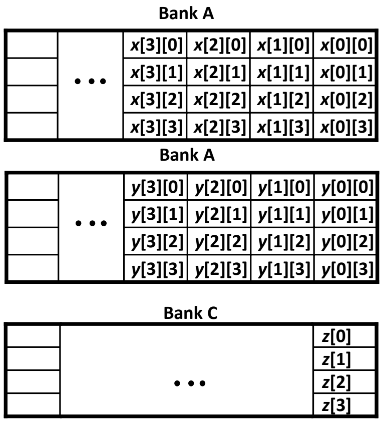
(a)

```
for (i = 0; i <= 3; i = i+1)
{
   t1 = x[i][0]+y[i][0] ;  ⎫
   t2 = x[i][1]+y[i][1] ;  ⎬ LD/+
   t3 = x[i][2]+y[i][2] ;  ⎪
   t4 = x[i][3]+y[i][3] ;  ⎭

   t1 = t1*c[0];           ⎫
   t2 = t2*c[2];           ⎬ ×
   t3 = t3*c[3];           ⎪
   t4 = t4*c[4];           ⎭

   tmp1 = t1+ t2 ;         ⎫ 1+
   tmp2 = t3+ t4 ;         ⎭

   z[i] = tmp1+ tmp2 ;     } 2+

   z[i] = K*z[i]           } ×/ST
}
```

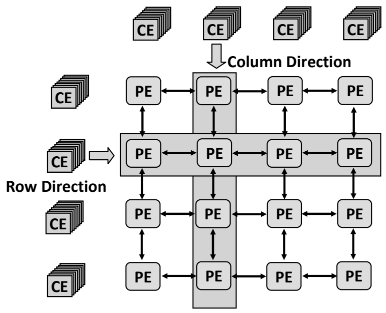
(b)

Fig. 3. C-code of Eq. (1). (a) Before parallelization, (b) After parallelization.

The SIMD-based scheduling enables parallel execution of multiple loop iterations as shown in Fig. 4(c), whereas the MIMD-based scheduling enables loop pipelining as shown in Fig. 4(d). The first row of Fig. 4(c) represents the direction of configuration broadcast. The second row of Fig. 4(c) and the first row of Fig. 4(d) indicate the schedule time in cycles from the start of the loop. In the case of SIMD model, load and addition operations in PEs are executed on all columns till 4th cycle with broadcast in column direction. Then the PEs in a row perform the same operation with broadcast in row direction. In the case of loop pipelining, PEs in the first column perform load and addition operations in the first cycle and then perform multiplications in the second cycle. In the next two cycles, the PEs in the first column perform summations, while the PEs in the next column perform multiplication and summation operations. When the first column performs the multiplication/store operation in the 5th cycle, the fourth column performs multiplication. Comparing the latency, SIMD takes three more cycles. As shown in this example, SIMD model does not utilize PEs efficiently since all data should be loaded before the computations of the same type are performed synchronously. On the other hand, since MIMD allows


Bank A
x[3][0] x[2][0] x[1][0] x[0][0]
x[3][1] x[2][1] x[1][1] x[0][1]
x[3][2] x[2][2] x[1][2] x[0][2]
x[3][3] x[2][3] x[1][3] x[0][3]
Bank A
y[3][0] y[2][0] y[1][0] y[0][0]
y[3][1] y[2][1] y[1][1] y[0][1]
y[3][2] y[2][2] y[1][2] y[0][2]
y[3][3] y[2][3] y[1][3] y[0][3]
Bank C
z[0]
z[1]
z[2]
z[3]

(a)



(b)

Broadcast	Column Direction				Row Direction			Column Direction			
Cycle Time	1	2	3	4	5	6	7	8	9	10	11
Column#1	LD/+	NOP	NOP	NOP	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	NOP	NOP	×	1+	2+	NOP	×/ST	NOP	NOP
Column#3			LD/+	NOP	×	1+	2+	NOP	NOP	×/ST	NOP
Column#4				LD/+	×	1+	2+	NOP	NOP	NOP	×/ST

(c)

Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	×	1+	2+	×/ST	NOP	NOP
Column#3			LD/+	×	1+	2+	×/ST	NOP
Column#4				LD/+	×	1+	2+	×/ST

Symbol	Meaning
LD/+	Data Load and Addition
NOP	No Operation
×	Multiplication
1+, 2+	Addition
×/ST	Multiplication and Store

(d)

Fig. 4. Execution model for CGRA. (a) Operand and result data in FB, (b) Configuration broadcast, (c) SIMD model, (d) Loop pipelining schedule.

any type of computations at any moment, it does not need to wait for a specific data to be loaded but can process other data that is readily available. Loop pipelining is an effective way of exploiting this fact, thereby utilizing PEs better. The loop pipelining in the example of Fig. 4 improves the performance by three cycles compared to the SIMD, but for loops with more frequent memory operations, it will have higher performance improvement.

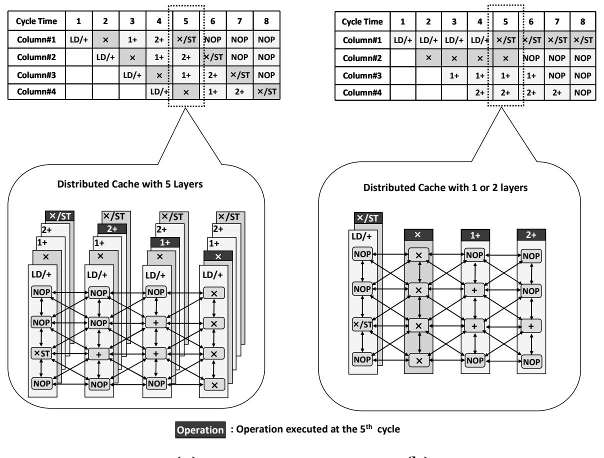
When mapping kernels onto the reconfigurable architecture with loop pipelining, we can consider two mapping techniques: spatial mapping and temporal mapping. Fig. 5 shows the difference between the two techniques with the previous example. In the case of temporal mapping (Fig. 5(a)), like the previous illustration of loop pipelining in Fig. 4(d), a PE executes multiple operations within a loop by changing the configuration dynamically. Therefore, complex loops having many operations with heavy data dependencies can be mapped better in temporal fashion, provided that the configuration cache has sufficient layers to execute the whole loop body.

In the case of spatial mapping, a loop body is spatially mapped onto the reconfigurable array implying that each PE executes a fixed operation with static configuration as shown in Fig. 5(b). The advantage of spatial mapping is that it may not need reconfiguration during execution of a loop. As can be seen from Fig. 5, spatial mapping needs only one or two cache layers whereas temporal mapping needs 4 cache layers. One disadvantage of spatial mapping is that spreading all the operations of the loop body over the limited reconfigurable array may require too many resources. Moreover, data dependencies between the operations should be taken care of by allocating interconnect resources to provide a path and inserting registers (or using PEs) in the path to synchronize the arrival of operands. Therefore, if the loop is simple enough to map the loop body to the limited reconfigurable array and there is not much data dependency between the operations, then spatial mapping is the right choice. The effectiveness of the mapping strategies depends on the characteristics of the target architecture as well as the target application.

IV. Motivation

1. Limitation of Existing Communication Structures

A typical coarse-grained reconfigurable architecture consists of a microprocessor, a Reconfigurable Array Architecture (RAA), and their interface. I can consider three types of organizations in connecting RAA to the processor. First, the array can be connected to the processor through a system bus as an 'Attached IP' [2, 3, 8] shown in Fig. 6(a). In this case, the main benefit of this organization is the ease of constructing such a system using a standard processor without modifying the processor and its compiler. In addition, large data buffer of RAA can be used to support applications having large inputs/outputs. However, the speed improvement using the RAA may have to compensate for significant communication overhead between the processor and RAA through system bus as well as SRAM-based large data buffer in RAA consumes much power. Second type of organization involves the array connected with the processor as a 'Coprocessor' [5-7] shown in Fig. 6(b). In this case, the standard processor does not change and the communication is faster than 'Attached IP' type interconnects because the coprocessor register-set is used as data buffer of the RAA and the processor can access the register-set by coprocessor data transfer instructions. In addition, the register-set consumes less power than the data buffer of 'Attached IP'. Since the size of the



Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	×	1+	2+	×/ST	NOP	NOP
Column#3			LD/+	×	1+	2+	×/ST	NOP
Column#4				LD/+	×	1+	2+	×/ST

Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	LD/+	LD/+	LD/+	×/ST	×/ST	×/ST	×/ST
Column#2		×	×	×	×	NOP	NOP	NOP
Column#3			1+	1+	1+	1+	NOP	NOP
Column#4				2+	2+	2+	2+	NOP

Distributed Cache with 5 Layers
Distributed Cache with 1 or 2 layers
Operation : Operation executed at the 5th cycle

(a) (b)

Fig. 5. Comparison between (a) Temporal mapping, (b) Spatial mapping.

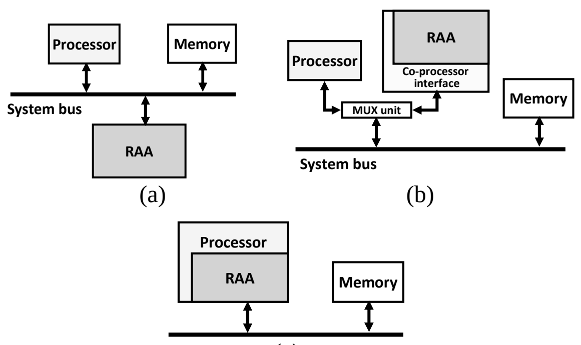


Fig. 6. Basic types of RAA coupling. (a) Attached IP, (b) Coprocessor, (c) Functional unit.

Table 1. Comparison of the basic coupling types

Coupling type	*Comm' power	**Comm' speed	Performance Bottleneck	Application feasibility
Attached IP	high	slow	communication through system bus	large size of input/output
Coprocessor	low	fast	limited size of coprocessor register-set	small size of input/output
Functional unit	low	very fast	limited size of processor registers	small size of input/output

* Comm' power: power consumption by data-storage (data buffer or registers)
** Comm' speed: Communication speed between processor and RAA

register-set is fixed by the processor ISA, it creates performance bottleneck for registers-PE array traffic due to applications having large inputs/outputs run on the RAA. In the third type of organization, the array is placed inside the processor like a 'FU (Functional Unit)' [4, 10, 11] as shown in Fig. 6(c). In this case, the instruction decoder issues special instructions to perform specific functions on the RAA as if it were one of the standard functional units of the processor. In this case, the communication speed is faster than 'Coprocessor' and power consumption of the data storage is less than 'Attached IP' because the processor register-set is used as data buffer of the RAA and the processor can directly access the register-set by the processor instructions. However, standard processor needs to be modified for due to integration with RAA and its compiler should be also changed. The performance bottleneck is caused by limited size of the processor registers as in the case of 'Coprocessor' type organization. Table 1 shows a summary about advantage and disadvantage of three coupling types.

2. RAA-based Computing Hierarchy

As mentioned in the previous section, basic three types of RAA organizations show advantage and disadvantage according to the input/output size of the applications. It shows the existing coupling structure with a conventional RAA cannot be flexible to support various applications with sacrificing performance. In addition, such an RAA structure cannot efficiently utilize PE arrays and data buffers leading to high power consumption.

I hypothesize that if CGRA can maintain a computing hierarchy of its RAAs having different size and communication speed, the CGRA-based embedded system can be optimized for its performance and power. It is because such a hierarchical arrangement of the RAA can optimize the communication latency and efficiently utilize functional resources of PE array in various applications. In this paper I propose a new CGRA-based architecture that supports such a RAA-based computing hierarchy.

V. COMPUTING HIERARCHY IN CGRA

In order to implement efficient CGRA-based embedded systems[1], I propose a new computing hierarchy consisting of two computing blocks using two types of coupling structures together – 'Attached IP' and 'Coprocessor'. In this organization, a general RAA having large size PE array is connected to a system bus and another is a small RAA composed of small PE array coupled with a processor through coprocessor interface. I call the small RAA *reconfigurable computing cache* (RCC) because it plays an important role in enhancing performance and power of the entire CGRA like data cache. The RCC and the RAA share critical resources and such a sharing structure provides efficient communication interface between two computing blocks. The proposed approach ensures that the RCC and the RAA are efficiently utilized to support variable size of inputs and outputs for variety of applications. In subsection V.1 and V.2, I describe computing hierarchy and resource sharing/pipelining in RCC and RAA in detail. Then I show how to optimize computing flow based on reconfigurable computing cache according to the applications in subsection V.3.

1. Computing Hierarchy – Size and Speed

A CGRA-based computing hierarchy is formed by splitting a conventional computing RAA block into two computing blocks – RCC with small PE array and RAA having large PE array as shown in Fig. 7(a). The RCC is coupled with coprocessor interface and the RAA is attached to a system bus as shown in Fig. 7(b). The RCC provides fast communication with the processor and

[1] I exclude the type of 'Functional Unit' from the proposed computing hierarchy because it requires the modification of processor and its compiler and entire CGRA is heavily dependent on the specific processor architecture. Our proposed approach aims to implement entire systems allowing any combination of standard processors and RAAs.

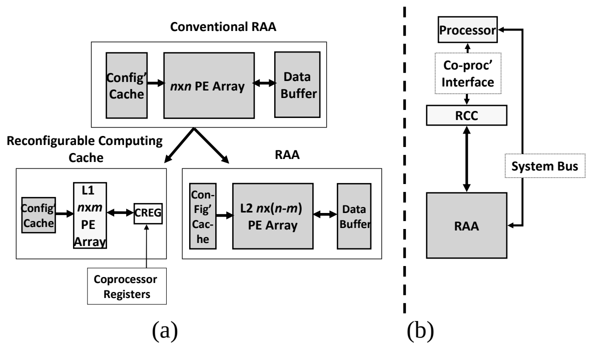


Fig. 7. Computing hierarchy of CGRA. (a) Size, (b) Speed.

offers low power consumption by using coprocessor register-set and small array size. Therefore the RCC can enhance performance and reduce power consumption when small applications run on CGRA. If RCC is not sufficient to support computing requirements of in applications, intermediate data from the RCC can be moved to the RAA through the interconnections as shown in Fig. 8. Such interconnections between the two blocks offer flexibility in migrating computing demands from one to another. Such computing flow may help to optimize performance and power for the applications having various sizes of inputs/outputs whereas the existing models show performance bottlenecks caused by the communication overheads or the limited size of the data-storages as shown in Table 1. I have described the computing flow optimization in detail in subsection V.3.

2. Resource Sharing and Pipelining in RCC and RAA

I have so far presented two factors (speed and size) in building computing hierarchy for CGRAs similar to memory hierarchy. It seems a small portion of RAA has been detached from large CGRA block and placed as the fast RCC block adjacent to the processor coupled with coprocessor interface. However, only considering two factors is not sufficient to design compact RCC for power and area benefits. This is because computing blocks can have diverse functionality which affects the system capabilities. The functionality of computing blocks is specified by functional resources of its PE such as adder, multiplier, shifter, logic operations etc. Therefore, it is necessary to examine how to select the functionalities of RCC and RAA. This leads to further studies on resource assignment between RCC and RAA. In the next subsection, I describe basic techniques to efficiently assign the resources between RCC and RAA - resource sharing and pipelining in CGRA [9]. Although these techniques are well known and widely used in other digital systems design, to the best of my knowledge, [9] is the first attempt to apply them to an RAA. Based on the basic techniques, I present an extension of resource sharing and pipelining in RCC and RAA in subsection V.2.2.

A. Resource Sharing and Pipelining

Fig. 9 shows the snapshot taken at the 5th cycle of execution of the previous example for three cases as shown in Fig. 5: (a) SIMD and two cases of loop pipelining - (b) temporal mapping and (c) spatial mapping. The operations in the 5th cycle for (a), (b) and (c) include multiplication and therefore the multipliers in the PE array are to be used. In the case of SIMD, all PEs perform multiplication requiring all of them to have multipliers, thereby increasing the area cost of the PE array. However, in the case of temporal mapping, only PEs in the 1st column and the 4th column perform multiplication while PEs in the 2nd and 3rd columns perform addition. In the spatial mapping, only PEs in the 1st and 2nd columns perform multiplication. As can be observed, in the temporal mapping and spatial mapping, there is no need for all PEs to have the same functional resources at the same time. This allows the PEs in the same column or in the same row to share area-critical resources. Fig. 10 shows four PEs in a row sharing two multipliers[2] at the 5th cycle in temporal mapping and

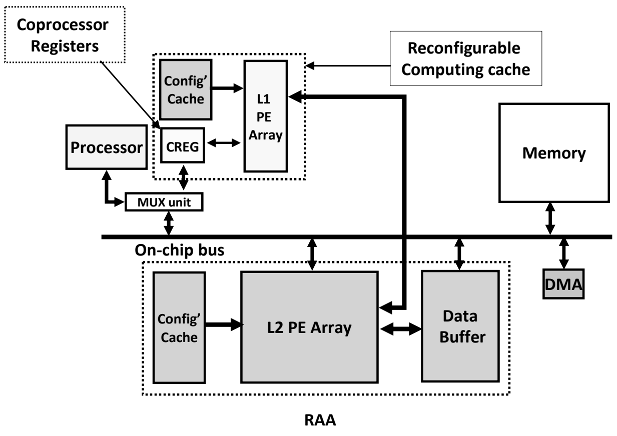
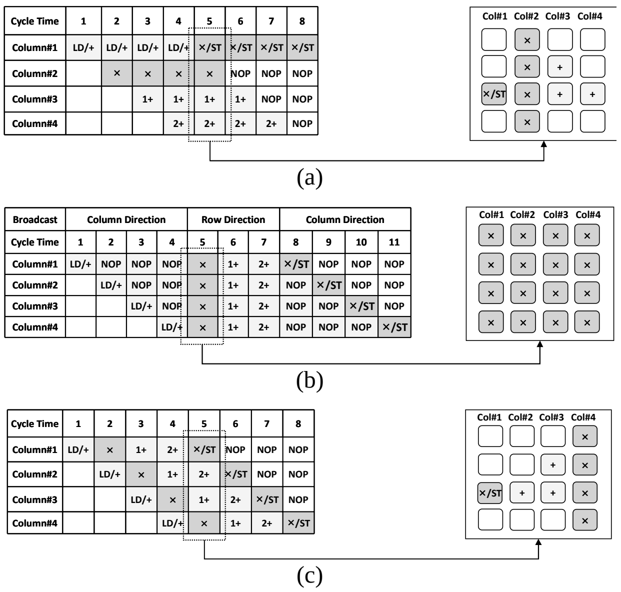


Fig. 8. CGRA configuration with RCC and RAA.

[2] Since multipliers take much more area than other resources, we classify them as critical resources.

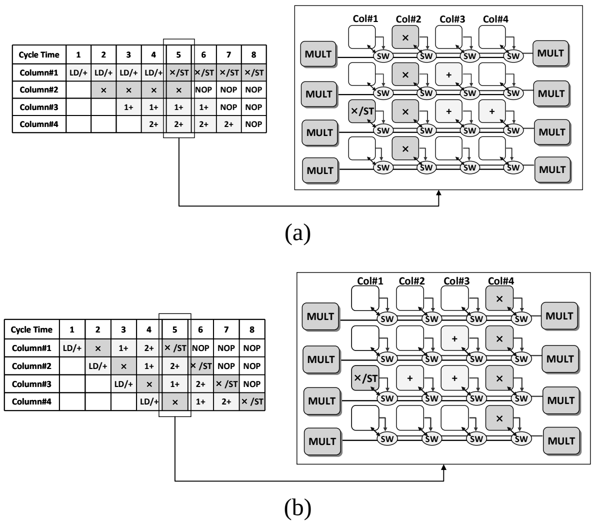


Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	LD/+	LD/+	LD/+	×/ST	×/ST	×/ST	×/ST
Column#2		×	×	×	×	NOP	NOP	NOP
Column#3			1+	1+	1+	1+	NOP	NOP
Column#4				2+	2+	2+	2+	NOP

Broadcast	Column Direction				Row Direction			Column Direction			
Cycle Time	1	2	3	4	5	6	7	8	9	10	11
Column#1	LD/+	NOP	NOP	NOP	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	NOP	NOP	×	1+	2+	NOP	×/ST	NOP	NOP
Column#3			LD/+	NOP	×	1+	2+	NOP	NOP	×/ST	NOP
Column#4				LD/+	×	1+	2+	NOP	NOP	NOP	×/ST

Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	×	1+	2+	×/ST	NOP	NOP
Column#3			LD/+	×	1+	2+	×/ST	NOP
Column#4				LD/+	×	1+	2+	×/ST

Fig. 9. Snapshots of three mappings. (a) SIMD, (b) Temporal mapping, (c) Spatial mapping.



Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	LD/+	LD/+	LD/+	×/ST	×/ST	×/ST	×/ST
Column#2		×	×	×	×	NOP	NOP	NOP
Column#3			1+	1+	1+	1+	NOP	NOP
Column#4				2+	2+	2+	2+	NOP

Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	×	1+	2+	× /ST	NOP	NOP	NOP
Column#2		LD/+	×	1+	2+	× /ST	NOP	NOP
Column#3			LD/+	×	1+	2+	× /ST	NOP
Column#4				LD/+	×	1+	2+	× /ST

Fig. 10. Eight multipliers shared by sixteen PEs. (a) Temporal mapping, (b) Spatial mapping.

spatial mapping. I depict only the connections related to resource sharing.

Fig. 11 depicts the detailed connections for multiplier sharing. The two n-bit operands of a PE are connected to the bus switch. The dynamic mapping of a multiplier to a PE is determined at compile time and the information is encoded into the configuration word. At run-time, the mapping control signal from the configuration word is fed to the bus switch and the bus switch decides where to route the operands. After the multiplication, the 2n-bit output is transferred from the multiplier to the original

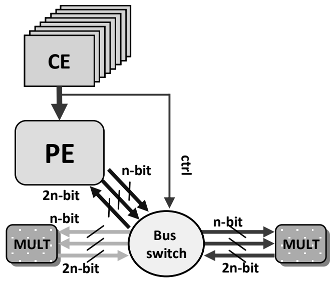


Fig. 11. The connection between a PE and shared multipliers.

issuing PE via the bus switch.

If there is a critical functional resource with long latency in a PE, the functional resource can be pipelined to curtail the critical path. Resource pipelining has clear advantage in loop pipelining execution because heterogeneous functional units with different delays can run at the same time. In the traditional design (Fig. 12(a)), the latency of a PE is fixed but in the pipelined PE design (Fig. 12(b)), we allow multi-cycle operations and so the latency can vary depending on the operation. This helps increase the system clock frequency.

If a critical functional resource such as a multiplier has both large area and long latency, the resource sharing and resource pipelining can be applied at the same time in such a way that the shared resource executes multiple operations at the same time in different pipeline stages. With this technique, the conditions for resource sharing are relaxed and so the critical resources are utilized more efficiently. Fig. 13 shows this situation. Through the pipelining, we can reduce the number of multipliers from 8 to 4 to perform the execution without any stall. This is because two PEs sharing one pipelined multiplier can perform two multiplications at the same time using different pipeline stages.

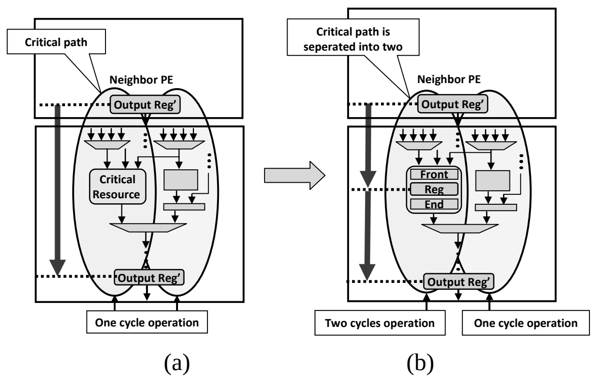
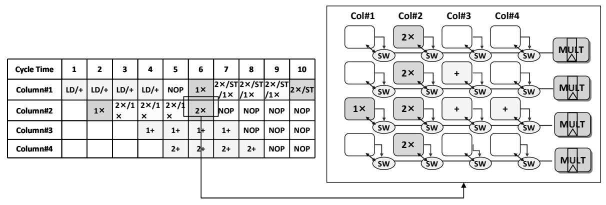
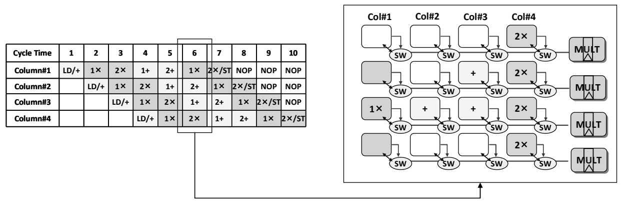


Fig. 12. Critical paths. (a) General PE, (b) Pipelined PE.



Cycle Time	1	2	3	4	5	6	7	8	9	10
Column#1	LD/+	LD/+	LD/+	LD/+	NOP	1×	2×/ST/1×	2×/ST/1×	2×/ST/1×	2×/ST
Column#2		1×	2×/1×	2×/1×	2×/1×	2×	NOP	NOP	NOP	NOP
Column#3				1+	1+	1+	1+	NOP	NOP	NOP
Column#4					2+	2+	2+	2+	NOP	NOP

(a)



Cycle Time	1	2	3	4	5	6	7	8	9	10
Column#1	LD/+	1×	2×	1+	2+	1×	2×/ST	NOP	NOP	NOP
Column#2		LD/+	1×	2×	1+	2+	1×	2×/ST	NOP	NOP
Column#3			LD/+	1×	2×	1+	2+	1×	2×/ST	NOP
Column#4				LD/+	1×	2×	1+	2+	1×	2×/ST

(b)

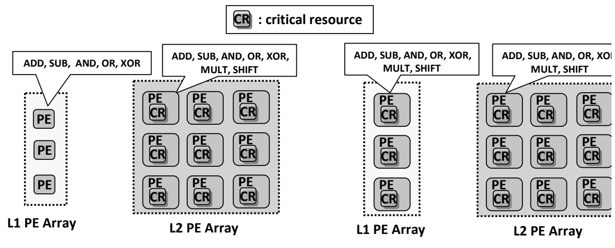
1×: First pipeline stage on multiplication, 2×: Second pipeline stage on multiplication

Fig. 13. Loop pipelining with pipelined multipliers. (a) Temporal mapping, (b) Spatial mapping.

B. Functional Resource Assignment Between L1 PE Array and L2 PE Array

First of all, I can classify the functional resources into two groups: primitive resources and critical resources. Primitive resources are basic functional units such as adder/subtractor and logical operators. Critical resources are area/delay-critical ones such as multiplier and divider. Based on the classification, let us consider two cases of the functional resource configurations as shown in Fig. 14. Fig. 14(a) shows hierarchical functionality that indicates L1 PE array has primitive resources and L2 PE array includes critical resources as well as primitive resources. The Fig. 14(b) shows identical functionalities both in the L1 and L2 PE arrays. In the case of Fig. 14(a), the RCC with L1 PE array is relatively lightweight computing block compared to the RAA with L2 PE array. Therefore, the RCC can perform small applications having only primitive operations with low power consumption. However, it causes 'lack of resource' problem when applications demand critical operations. In Fig. 14(b) L1 and L2 PE arrays have identical functionality with area and power overheads.

To prevent such extreme cases, I propose resource sharing and pipelining for the RCC and the RAA based on subsection V.2.1. L1 and L2 PE array have the same primitive resources and shared the pipelined critical resources as shown in Fig. 15. Here the RCC and the



(a) (b)

Fig. 14. Two cases of functional resource assignment. (a) Hierarchical functionality, (b) Identical functionality.

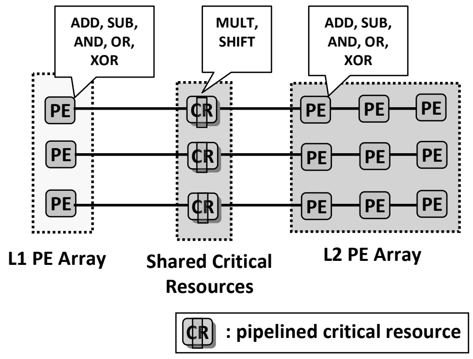
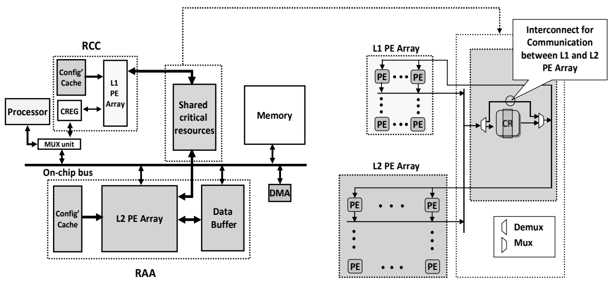


Fig. 15. Critical resource sharing and pipelining in L1 and L2 PE Array.

RAA basically perform the primitive operations and their functionality will include the critical operations using the shared resources. Fig. 16 shows interconnection structure with shared critical resources along with RCC and RAA. PEs in the same row of the L1 nd L2 array share the pipelined critical resources in the same manner as shown in subsection V.2.1. Such a structure avoids the 'lack of resource' problem in Fig. 14(a) and this structure is more area and power-efficient than Fig. 14(b) because the number of critical resources is reduced and the critical resources taken out of L1 and L2 PE array are not



(a) (b)

Fig. 16. Interconnection structure among RCC, shared critical resources and L2 PE Array. (a) Entire structure, (b) Interconnection structure.

affected by unnecessary switching activity caused by other resources. In addition, interconnections for resource sharing can be also utilized for communication interface between the RCC and the RAA by adding multiplexer and de-multiplexer between front and end of the critical resources as shown in Fig. 16(b).

C. Computing Flow Optimization

Based on the proposed CGRA structure, I can classify four cases of optimized computing flow to achieve low power and high performance. Fig. 17 shows such four computing flows on the proposed CGRA according to variance of input and output size of applications – In subsection V.1.2, Table 3 shows that I can select the optimal case among the proposed computing flows for several applications with variance in their input/output size. All of the cases show that shared critical resources are used as needed because they are only utilized when applications have the operations requiring the critical resources. Fig. 17(a) shows computing flow when application has the smallest inputs and outputs. In this case, only RCC functional units are used to execute the application while the RAA is disabled to reduce power consumption. However, if the application has larger inputs and outputs than Fig. 17(a), the computing flow can be extended to L2 PE array as shown in Fig. 17(b). Even though L2 PE array is used for this case, data buffer of the RAA is not used because the coprocessor register-set (CREG) is sufficient to save the all of the inputs or outputs. The next case is that when RAA is used with RCC because of large inputs and small outputs as shown in Fig. 17(c). In this case, data buffer of the RAA receives inputs using DMA which is more efficient for overall performance than CREG. This is because insufficient CREG resource for large inputs causes performance bottleneck with heavy registers-PE array traffic. Therefore, the L2 PE array may be used first for running such application and the L1 PE array can be utilized for enhancing parallelized execution as needed. However, the outputs are stored on CREG because their size is small. Finally, Fig. 17(d) shows a case of RAA used with L1 PE array with large inputs and outputs. To avoid heavy registers-PE array traffic by the large input/output size, the data buffer with DMA is used and L1 PE array can be optionally utilized for enhancing parallelized execution. In summary, the computing flow on the proposed CGRA can be adapted according to the input/output size of applications. It is more power-efficient than using a conventional CGRA by separated computing blocks with sharing critical resources. This way is only necessary computing blocks are utilized. In addition, computing flow with supporting two communication interfaces reduces power and enhances performance.

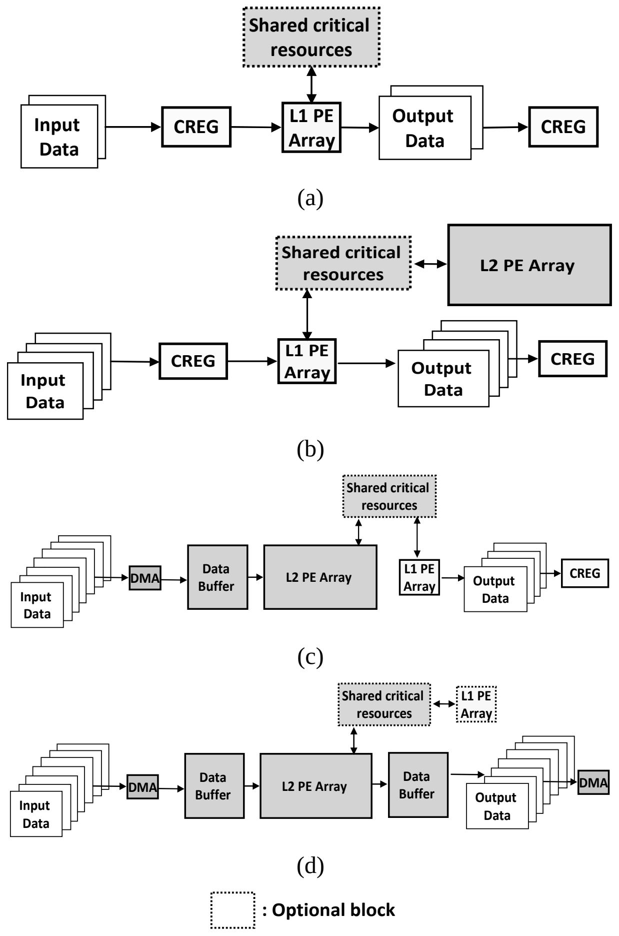


Fig. 17. Four cases of computing flow according to the input/output size of application. (a) Smallest inputs and outputs (STIO), (b) Small inputs and outputs (SIO), (c) Large inputs and small outputs (LISO), (d) Large inputs and outputs (LIO).

VI. EXPERIMENTS AND RESULTS

1. Experimental Setup

A. Architecture Implementation

To demonstrate the effectiveness of the proposed

Table 2. Comparison of the basic coupling types

CGRA	PE array	Data storage
Attached IP	8 × 8 PE array	6 KB data buffer
Coprocessor	8 × 8 PE array	256-byte coprocessor register-set
Proposed RCC-based	8 × 2 L1 PE array and 8 × 6 L2 PE array	4 KB data butter and 256-byte coprocessor register set

(Leon2 processor [15] is used as main processor)

RCC-based CGRA, we have designed three different organizations of CGRA with RT-level implementation using VHDL as shown in Table 2.

In addition, for resource sharing of RCC-based CGRA, two pipelined multipliers and two shifters are shared by PEs in the same row of L1 and L2 PE array whereas conventional two types of CGRA do not support such a resource sharing and pipelining.

The architectures have been synthesized using Synopsys Design Compiler with TSMC 0.18 μm technology. Synopsys PrimePower tools have been used for gate-level simulation and power estimation. To obtain the power consumption data, we have used the applications in Table 2 for simulation with operation frequency of 70 MHz and typical case of 1.8 V Vdd and 27 ℃.

B. Evaluated Applications

Evaluated applications are composed of real multimedia applications and benchmarks. We have analyzed the input/output size and operation-types in the applications to identify specific computing flow in Fig. 17. Table 3

Table 3. Application characteristics

Applications	SR	Flow	Benchmarks	SR	Flow
(H.263) 8 x 8 DCT	✓	SIO	*256-point FFT	✓	LISO
(H.263) 8 x 8 IDCT	✓	SIO	*256-tap FIR	✓	LISO
(H.263) 8 x 8 QUANT	✓	SIO	*Complex Mult	✓	LISO
(H.263) 8 x 8 DEQUANT	✓	SIO	**State	✓	STIO
(H.263) SAD	-	LISO	**Hydro	✓	STIO
(H.264) 4 x 4 ITRANS	✓	STIO	**Tri-Diagonal	✓	LIO
(H.264) MSE	✓	LISO	**First-Diff	-	STIO
(H.264) MAE	-	LISO	**ICCG	✓	STIO
(H.264) 16 x 16 DCT	✓	LISO	**Inner Product	✓	LIO
8 x 8 * 8 x 1 Matrix-Vector Multiplication	✓	SIO	*: DSPstone benchmarks [13] **: Livermore loop benchmarks [14] SR:'✓'means critical resources are used for the application. STIO: smallest inputs and outputs SIO: small inputs and outputs LISO: large inputs and small outputs LIO: large inputs and outputs		
16 x 16 * 16 x 1 Matrix-Vector Multiplication	✓	LISO			
8 x 8 Matrix Multiplication	✓	SIO			
16 x 16 Matrix Multiplication	✓	LISO			

shows the selected applications and the optimal computing flows for them.

I have used automatic compilation flow [17] to map applications onto the three architectures for supporting loop pipelining [16]. Binary context words are automatically generated from the compiler for temporal mapping. The timing and control information that is used to operate execution controller is manually optimized and the final encoded data is loaded onto registers of the execution controller.

2. Results

A. Area Cost Evaluation

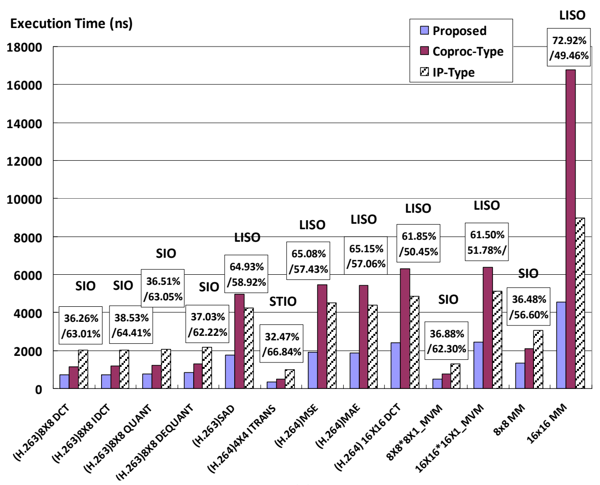
Table 4 shows area cost evaluation for the two cases. 'Base 8 × 8' means 8 × 8 PE array included in 'Attached IP' and 'Coprocessor' type CGRA. 'Proposed' means L1 and L2 PE array included in the proposed RCC-based CGRA. Even though interconnection area of the proposed model increases because of resource sharing structure, entire area of the proposed one is reduced by 22.68% because it has less critical resources than base 8 × 8 PE array.

Table 4. Area cost comparison

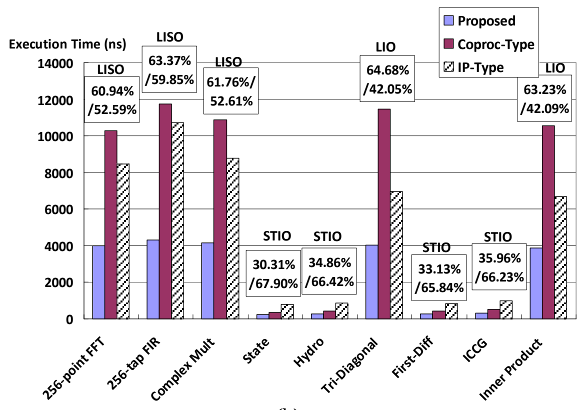
PE Array	No' of PEs	No' of MULTs	No' of SHTs	Gate Equivalent			R(%)
				Interconnect	Logic	Total	
Base 8 × 8	64	64	64	151234	478908	630143	-
Proposed	64	16	16	161311	325908	487219	22.68

B. Performance Evaluation

The synthesis results show that the proposed PE array has reduced critical path delay (5.12 ns) compared to the base PE array (8.96 ns). This is because pipelined multipliers are excluded from the original set of critical paths. Based on the synthesis results, we evaluate execution times of the selected applications on three cases of CGRA as shown in Fig. 18. The execution times are cycle–accurate because they have been obtained by running applications with designed RTL code on a HDL simulator. They include communication time between memory/processor and the RAA or RCC. Each application is executed on the RCC-based CGRA in the manner of selected computing flow as shown in Table 3



(a)



(b)

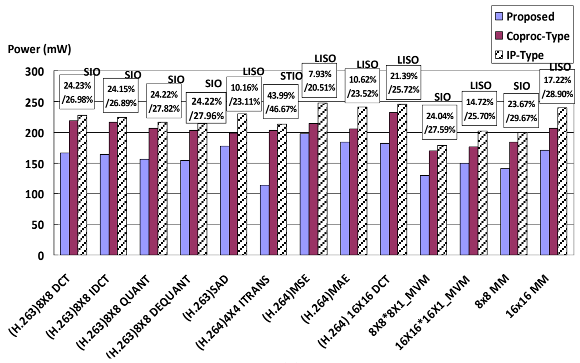
A%/B%: A% means reduced execution time ratio compared with Coproc-Type and B% means reduced execution time ratio compared with IP-Type.

Fig. 18. Performance comparison. (a) Real applications, (b) Benchmarks.

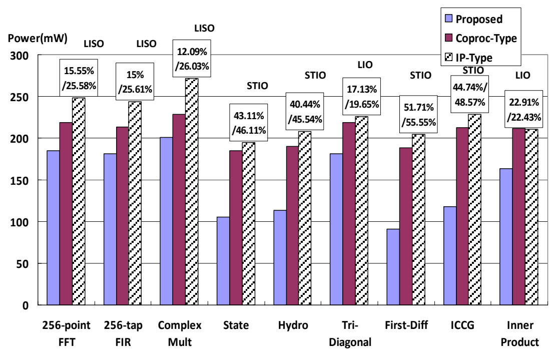
– all of the applications are classified under 4 cases of computing flow (STIO, SIO, LISO and LIO). In the case of STIO and SIO, performance improvement compared with ‘Coprocesssor’ type is relatively less (30.31%~37.03%) than LIO and LISO (60.94%~72.92%). This is because the improvements of STIO and SIO are achieved by only reduced critical path delay whereas the improvements of LIO or LISO are achieved by avoiding heavy coprocessor registers-PE array traffic as well as reduced critical path delay. However, compared with ‘Attached-IP’ type, STIO and SIO achieve much more performance improvement (56.60%~67.90%) whereas LISO and LIO show the improvement of (42.05%~59.85%). This is because STIO and SIO do not use data buffer of the RAA causing communication overhead on system bus.

C. Power Evaluation

Fig. 19 shows the comparison of power consumptions in three different organizations of CGRA. First of all, the proposed L1 and L2 PE array is more power-efficient than the base PE array because of the reduced critical resources. With such a power-efficient PE array, the amount of power saving depends on the selected computing flow for the application. The most power-efficient computing flow is STIO that shows relatively much power saving (40.44%~55.55%) compared to other cases (7.93%~29.67%) because the STIO does not use the RAA - specially, ‘First_Diff’ shows the highest power saving ratio of 51.71%/55.55% because of not using the shared critical resources. The next power-efficient model is SIO showing power saving (23.67%~29.67%). This is because the SIO computing flow does not use data buffer of the RAA whereas LISO (7.93%~26.03%) and LIO (17.13%~22.91%) utilizes the data buffer for input data or output data. Finally, power saving of LISO and LIO is mostly achieved by reduced critical resources and by not activating L1 PE array.



(a)



(b)

A%/B%: A% means power saving ratio compared with Coproc-Type and B% means power saving ratio compared with IP-Type.

Fig. 19. Power comparison. (a) Real applications, (b) Benchmarks.

VII. CONCLUSIONS

Coarse-grained reconfigurable architectures have emerged as a suitable solution for embedded systems because it aims to achieve high performance and flexibility. However, the CGRA has been considered as prohibitive one due to its significant area/power overhead and performance bottleneck. In addition, fixed communication structure between the computing block and processor can not guarantee good performance for various applications. To overcome the limitations, in this paper, I proposed a new computing hierarchy consisting of two reconfigurable computing blocks with two types of communication structure together. In addition, the two computing blocks have shared critical resources. Such a sharing structure provides efficient communication interface between them with reducing overall area. Experiments with implementation of several applications on the new hierarchical CGRA demonstrate the effectiveness of my proposed approach. The proposed approach reduces area by up to 22.68%, execution time by up to 72.92% and power by up to 55.55.% when compared with the existing undivided arrays in CGRA architectures.

ACKNOWLEDGMENTS

This Research was supported by the Sookmyung Women's University Research Grants 1-1103-0728.

REFERENCES

[1] Reiner Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in *Proc. of Design Automation and Test in Europe Conf.*, pp.642-649, Mar., 2001.

[2] Hartej Singh et al, “MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applica-tions,” *IEEE Trans. on Computers*, Vol.49, No.5, pp.465-481, May, 2000.

[3] A. Deledda et al, “Design of a HW/SW communication infra-structure for a heterogeneous reconfigurable processor,” in *Proc. of Design, Automation, and Test in Europe Conf.*, pp.1352-1357, Mar., 2008.

[4] C. Arbelo et al, “Mapping Control-Intensive Video Kernels onto a Coarse-Grain Reconfigurable Architecture: the H.264/AVC deblock-ing filter,” in *Design Automation and Test in Europe Conf.*, pp.642-649, Mar., 2007.

[5] T. Miyamori and K. Olukotun, “A quantitative analysis of reconfigurable coprocessors for multimedia appli-cations,” in *Proc. of IEEE Symp. on FPGAs for Custom Computing Machines*, pp.15-17, Apr., 1998.

[6] Michalis D. Galanis et al, “Speedups in embed-ded systems with a high-performance coprocessor datapath,” *ACM Transactions on Design Automation of Electronic Systems*, Vol.12, No.35, Aug., 2007.

[7] Timothy J. Callahan et al, “The Garp architecture and C compiler,” *IEEE Computer*, Vol.33, No.4, pp.62-69, Apr., 2000.

[8] A. S. Y. Poon, “An Energy-Efficient Reconfigurable Baseband Processor for Wireless Communications,” *IEEE Trans. on Very Large Scale Integration Systems*, Vol.15, No.3, pp.319-327, Mar., 2007.

[9] Yoonjin Kim et al, “Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization,” in *Proc. of Design Automation and Test in Europe Conf.*, pp.12-17, Mar., 2005.

[10] Francisco Barat et al, “Low power coarse-grained reconfigurable instruction set processor,” in Proc. of Int. Conf. on Field Pro-grammable Logic and Applications, pp.230-239, Sep., 2003.

[11] Marco Lanuzza et al, “Cost-effective low-power processor-in-memory-based reconfig' datapath for multimedia applications,” in *Proc. of Int. Symp. on Low Power Electronics and Design*, pp.161-166, Aug., 2005.

[12] Huesung Kim et al, “Low-power high-performance reconfigurable computing cache architectures,” *IEEE Trans. on Computers*, Vol.53, No.10, pp.1274-1290, Oct., 2004.

[13] http://www.ert.de/Projekte/Tools/DSPSTONE

[14] http://www.netlib.org/benchmark/livermorec

[15] Gaisler Research; http://www.gaisler.com/cms

[16] Jong-eun Lee et al, “Mapping loops on coarse-grained reconfigurable architectures using memory operation sharing,” in *Technical Report 02-34*, Center for Embedded Computer Sys-tems(CECS), Univ. of California Irvine, Calif., 2002.

[17] Jonghee W. Yoon et al, "Temporal Mapping for Loop Pipelining on a MIMD style Coarse-Grained Reconfigurable Architecture," in *Proc. of International SoC Design Conference*, Oct., 2006.

Yoonjin Kim received the B.S. degree in information and communication engineering from Sungkyunkwan University, Seoul, South Korea, in 2003, the M.S. degree in electrical engineering and computer science from Seoul National University, Seoul, South Korea, in 2005, and the Ph.D. degree in computer engineering from Texas A&M University, College Station, in 2009. From 2009 to 2010, he was a Senior R&D Staff Member with the Samsung Advanced Institute of Technology (SAIT), Gyeonggi, South Korea. Since 2010, he has been an Assistant Professor with the Department of Computer Science at Sookmyung Women's University in Seoul, South Korea. His research interests include embedded systems, computer architecture, VLSI/system-on-chip design, and hardware/software co-design.