A New Coarse-Grained Reconfigurable Architecture with Fast Data Relay and Its Compilation Flow

Lu Wan Chen Dong Deming Chen
ECE, University of Illinois at Urbana-Champaign
{luwan2, cdong3, dchen}@illinois.edu

Abstract

In this paper we propose a *Fast Data Relay* (*FDR*) mechanism and the supporting compiler techniques to enhance existing *CGRA*. Our results on *FDR*-based *CGRA* are compared with two other works in this field: ADRES and RCP. Experimental results for various multimedia applications show that FDR combined with a routing based resource conflict solver enables us to deliver up to 29% and 21% higher performance than ADRES and RCP, respectively.

1. Introduction and Motivation

A *Coarse-Grained Reconfigurable Architecture* (*CGRA*) can potentially provide higher performance for video and signal-processing applications than fine-grained *Field-Programmable Gate Arrays* (FPGAs). Two interesting CGRAs are ADRES and RCP. ADRES exploits a point-to-point communication mechanism and resolves resource conflict at compile time using modulo scheduling [1]. RCP proposes an architecture with ring-shaped buses and connection boxes for concurrent computation and communication [2]. We enhance the existing CGRA by proposing a *Fast Data Relay* (*FDR*) mechanism to enable concurrent multi-cycle communication between *Processing Elements* (*PEs*) with high efficiency. FDR can provide a fast data link between PEs so that (1) a corner-to-corner transmission in a tile can be finished in two cycles; (2) data communication can be done as a background operation without disturbing computation; and (3) source operands can have multiple copies in different PEs so that a dependent PE can find a local copy in its vicinity. In the following sections, we call the proposed architecture *FDR-CGRA*. To enable FDR, we introduce bypassing registers for PEs and propose the use of horizontal and vertical companion channels for the communication within a tile. To utilize FDR efficiently, we also propose two compiler techniques inspired by circuit design automation to map kernel code onto CGRA: (1) Decouple operation scheduling into two phases: placement and routing. Placement puts operations on critical paths close to one another, while routing finds single-hop communication paths for critical operations and multi-hop communication paths for noncritical operations through FDR. (2) Instead of resolving resource conflict during modulo scheduling, our resource constraints are modeled as routing constraints and resolved explicitly during routing. This could avoid the underutilized wind-up and wind-down phases in modulo scheduling for certain applications.

Our tools are also parameterizable to allow further architecture design exploration. In the following sections, we will first present the architectural support for FDR and then introduce routing-based compiler support for the efficient use of FDR. The experimental results will be discussed at last.

2. Architecture Overview

As with existing reconfigurable computing platforms surveyed in [7], we assume that a host CPU initializes the kernel computation on our FDR-CGRA and is in charge of system-level dataflow via DMA. Thus FDR-CGRA can focus on accelerating time-consuming kernels. Figure 1 provides a conceptual illustration of the overall FDR-CGRA system. In FDR-CGRA, the basic computation units are PEs (Section 3.1) organized in tiles, and a wire-based communication layer (Section 3.2) glues PEs together.

FDR-CGRA adopts the tile-based design philosophy, like that of existing commercial FPGAs, for design scalability. Considering that memory banks have limited bandwidth, we restrict that only the shadowed PEs can access memory banks with load/store instructions.

To exploit more instruction level parallelism, it is desirable to exchange data for operations on noncritical paths without intervening with critical computation. Fast Data Relay is such a mechanism. It is capable of routing data from one PE to another in multiple cycles in parallel with computation by utilizing the bypassing registers and companion channels. It is an enhancement over ADRES's point-to-point computation. The philosophy of concurrent computation and communication was exploited in the RAW microprocessor [3]. However, instead of using switches that can route data dynamically, FDR-CGRA uses wire-based channels, because it is an application-specific accelerator. Thus, communication within a tile is faster in FDR-CGRA than in RAW. The detailed analysis can be found in Section 3.2.

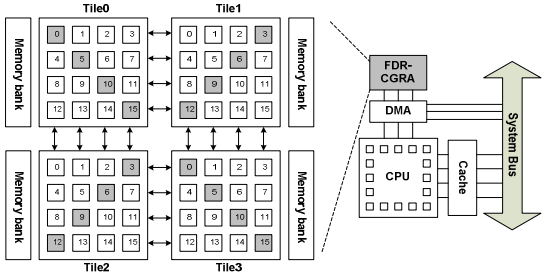

Figure 1: FDR-CGRA system overview

3. Hardware Support

At the hardware level, both the PE design and the communication mechanism need to be modified to enable FDR.

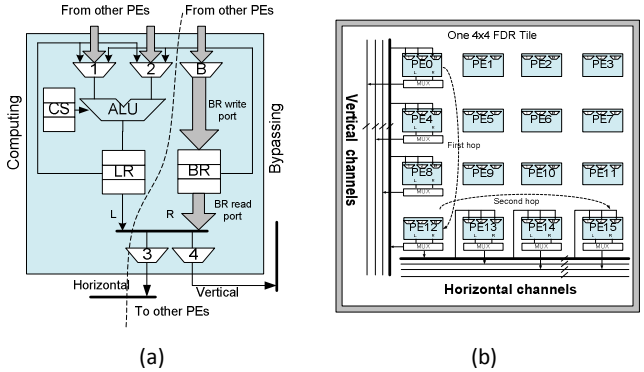

Figure 2: (a) Architectural support for Fast Data Relay at the PE level, (b) Companion channels and Fast Data Relay

3.1 Processing Element and Bypassing Registers

As shown in Fig. 2(a), each PE has a dedicated computation data path (left) and a communication bypassing path (right) that can be used for FDR. This dedication is essential to enable simultaneous computation and communication. The ALU is driven by traditional assembly code and its functionality can be configured dynamically

through the *Context SRAM* (*CS*). The CS stores both the function specification of the ALU and the communication information that provides cycle-by-cycle reconfiguration of channels among PEs. The communication information includes control bits for the input multiplexers (1,2,B) on the top of the PE and output multiplexers at the bottom (3,4).

Another key feature of the enhanced PE is the *Bypassing Register File* (*BR*): in addition to the normal *Local Register File* (*LR*), which is used to store intermediate data, PE is enhanced by including the BR, through which multi-cycle data communication can be done. The BR can store and hold the intermediate routing data until it is fetched by downstream PEs. These BRs provide more programming flexibility over the point-to-point communication used in ADRES. However, at the same time the BR poses some difficulty for our compiler. This will be discussed in Section 4.2.

3.2 Wire-Based Companion Channels

In FDR-CGRA, inter-tile communication is simply through direct connections between PEs on the tile boundaries, as shown in Fig. 1, while intra-tile communication is implemented with companion channels (Fig. 2(b)). Each PE is associated with two 32-bit companion channels: vertical and horizontal. The vertical (horizontal) channel is used to send scalar data from the sender PE to any PE in the same column (row). Note that vertical/horizontal channels span only across the PEs within the same tile for scalability reasons. Initialized by the sender PE, scalar data can be deposited onto either horizontal or vertical channel through the output multiplexer. At the second half of the same cycle, the receiver PE can fetch the scalar data from its corresponding input multiplexer. The control bits of the output multiplexer and input multiplexer are determined at compile time and stored in the Context SRAM.

Multi-cycle communication in processor array was pioneered in RAW [3]. However, in contrast to the full-blown 8-pipeline-stage MIPS processor used in RAW, each PE in our architecture is a lightweight, simple processor consisting of three pipeline stages - decode, execute, and write-back - with the goal of accelerating the program kernel. We also assume that data dependency in assembly code can be determined at compile time, which is true in general for kernel code in video and signal-processing applications. Thus resource-demanding dynamic switches, used in RAW and Network-on-Chip, can be avoided in our design, resulting in a wire-based, simple communication mesh. This simplification also allows PE and communication infrastructure to be implemented with a small footprint. ADRES [1] assumed that, in a single hop (a single cycle), a signal can travel across 5 PEs. RCP assumed that one hop can travel across 3 PEs, because of the complexity of its crossbar-like *Connection Box* design [2]. Given the simplicity of our architecture, we assume that one hop of data transmission can travel across 4 PEs. As a result, our wire-based companion channels enable a corner-to-corner transmission in a 4×4-tiled FDR architecture to finish (but not necessarily) in 2 cycles (PE0→PE12→PE15) as shown in Fig 2(b). In contrast, RAW needs 6 hops to go from corner to corner [3] because it uses sophisticated MIPS processors. Because FDR-CGRA is an application-specific accelerator, in our experimental results we compare it only with two representative works in this field: ADRES and RCP.

All PEs on the multi-cycle communication path except the sender PE are called *linking PEs*. During the multi-cycle FDR, the source scalar data will be transmitted to a receiver PE through linking PEs. Along the transmission, all linking PEs on the routing path will have a local copy of the origin data. These local copies can be either discarded immediately after fetched by the first dependent operation (*Volatile Copy*) or kept for future reference (*Nonvolatile Copy*). Later on, if an operation on another PE has dependency on this origin data, instead of getting it from the origin PE, it could possibly find a local copy in its vicinity from a linking PE. This can effectively detour traffic from the origin PE to some non-congested linking PEs. We call those linking PEs that provide local copies *identical start points* in the routing. Whether to keep the scalar data in linking PEs' BR as a volatile copy or as a nonvolatile copy is decided at the routing stage.

4. Supporting Compiler Techniques

The operation mapping problem is formulated as: Given the *Directed Acyclic Graph* (DAG) representing the data dependency among operations in the kernel code, find a valid scheduling and mapping of all operations on the *routing region* so that all data dependency is satisfied and no resource conflict exists, where the routing region is constructed as shown in Fig. 3 by the following steps: (1) Line up all PEs (Fig. 3(a)) in the two-dimensional computation grid along the *x* axis. The PEs on the grid are marked as *PE(x,y)*, where *x* is the PE's ID and *y* means this *PE* is at schedule step *y*. (2) Duplicate the PE row as many times as the minimum feasible (regardless of the detailed routing congestion) scheduling steps along the time axis,. (3) Add edges between any nearby PE rows to reflect the connectivity among PEs according to the connectivity specification. Figure 3(b) shows a sample routing region consisting of 4 PEs. Figure 3(c) is a DAG representing data dependency among operations. (1) Solid edges represent single-cycle data dependency, and (2) the dashed edge represents multi-cycle dependency, which may require FDR. Our operation mapping consists of two phases: placement (*frontend*) and routing (*backend*). The placement phase only suggests a trial operation mapping onto slots in the routing region and leaves the detailed resource conflict problems to be resolved later in the routing phase to get a final mapping of operations to routing region as shown in Figure 3(d).

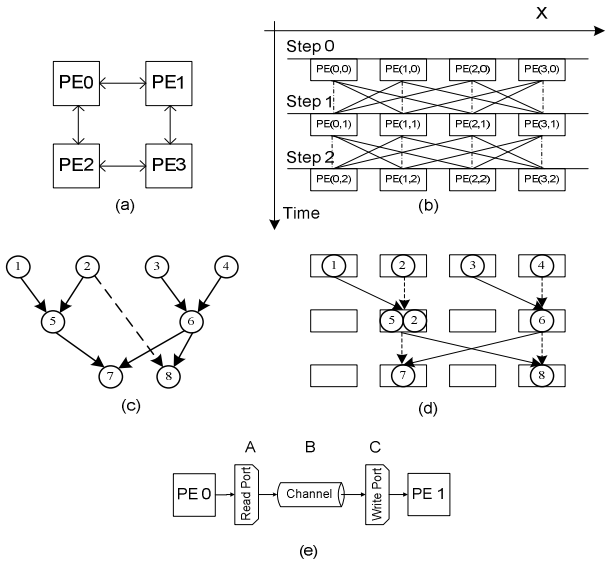

Figure 3: (a) A 2×2 PE array, (b) the routing region, (c) a DAG, (d) the mapping and routing solution, (e) constraint model

In contrast to the modulo scheduling used in the mesh-like CGRA compilers [1][2][4], our approach could achieve performance saturation by removing the *Iteration Interval* (*II*) constraints [1]. Although modulo scheduling is commonly used to optimize loops with arbitrary large number of iterations, the performance could suffer from the un-saturation of operations during the wind-up and wind-down phases. This problem is particularly severe for a compact loop body with small number of iterations. Unfortunately, many applications demonstrate this property because of the popularity of partitioning large kernels into small pieces for data localization. Our two-phase procedure can virtually remove the II constraints by unrolling the whole loop and explicitly resolving resource conflicts during the backend data routing stage. Considering that the routing algorithm in circuit design can typically handle hundreds of thousands of instances, this dedicated routing phase enables us to work on the fully unrolled kernel loops.

4.1 Placement frontend

Since the placement frontend only suggests a trial operation mapping and leaves the detailed resource conflict problems to be resolved later in the backend routing stage, virtually any scheduling algorithm can be used in the frontend stage, including modulo scheduling. We find from experiment that even the simple and fast *list scheduling* can give a fairly good initial placement. We need to point out that: although the detailed resource conflict will be resolved in backend routing stage, the frontend still tries to avoid creating difficulties for routing. The placer is based on a list scheduling algorithm guided by operation criticality so that critical operations will be placed on PEs that are directly linked with companion channels and noncritical operations will be placed on PEs without direct links.

4.2 Routing backend

All the resources, including channels and register ports, are assigned capacities. The channels and register ports involved in FDR are associated with a cost, which is dynamically calculated during the iterative routing procedure to reflect the quality of the current solution. The global routing algorithm [5] iteratively improves the quality of the solution by detouring the routes in the congested area to less congested areas in the routing region until an acceptable solution is achieved. After introducing the constraint model, we will discuss some specific techniques important to FDR during the routing.

Resource constraints: We apply channel capacities on vertical and horizontal companion channels and apply an upper bound on the number of total simultaneous accesses to the BR files in PEs to reflect the bandwidth constraint of a physical register file (2R2W). Each segment of a communication link is modeled as a cost function of three parts (Fig. 3(e)): read port, channel, and write port. The cost is proportional to the congestion on this communication link. Whenever a channel (port) is used, the capacity of the channel (port) will decrease and the cost of using the link will increase accordingly.

Conflict solver: Rip-up and reroute come in place to resolve congestion whenever resource conflict exists. Given an initial congested solution, rip-up and reroute will perform the following steps: (1) Pick a routed path, usually a path that can afford extra latency without degrading overall performance; (2) rip-up (or break) the path by deallocating all the resources (i.e., channels and ports) taken by this path except the source PE and the destination PE; (3) based on the existing congestion information of the routing region, find an uncongested detour and reroute the path. These steps are iterated for other paths until the congestion is eliminated. The reroute is guided by a cost function. Rip-up and reroute stops when no improvement can be achieved. Fig 3(d) is a routed design; circles represent operations, rectangles represent PEs, a solid line means a companion channel is used for communication between PEs, and a dashed arrow means that the PE has a data dependency on itself. Considering the dependency 2→8 in the DAG in Fig. 3(c), 2 is mapped onto PE1 and 8 is mapped onto PE3. We need send the data from PE1 to PE3 in two cycles. Two possible paths exist: PE1→PE3→PE3 or PE1→PE1→PE3. If the conflict solver cannot resolve the resource conflict on the first path, it will then rip-up that path and use the alternative one.

Using non-volatile copy: During the routing, linking BR could be (but not necessarily) used as *identical start point* for routing thereafter. This is equivalent to having multiple instances of a value at different locations in the routing region; thus the chance to find an uncongested path from nearby PEs greatly increases. At the beginning of a routing iteration, all storages in BR are reclaimed. If a path is ripped up, then all BRs used in this path are reclaimed. If no congestion exists in the last iteration, we keep all scalar data that are actually used as identical start points stored in BRs as nonvolatile copies, and all other scalar data in BRs are treated as volatile copies.

Schedule step relaxation: If *conflict solver* and *using non-volatile copy* cannot resolve all the congestions, *schedule step relaxation* (*SSR*) will be invoked to insert extra *Scheduling Steps* into the most congested regions. By interleaving rip-up and reroute and *SSR*, an un-congested feasible solution is guaranteed.

5. Experimental Results

Several programs are extracted from open source applications as benchmarks, including idct, interpolation, SAD16 from xvid 1.1.3 and various get_block functions from JM7.5b. These C/C++ benchmark programs are first preprocessed with *Low Level Virtual Machine* (*LLVM*) [6]. The compiled kernel code is processed by LLVM for constant propagation and loop unrolling. Our operation mapping algorithm takes the asm code produced by LLVM and performs the placement and routing to map operations onto FDR-CGRA. To preserve data dependency, we implemented a parser to interface between the virtual instructions and our algorithm. The validation of our approach is done by analyzing the dumped execution trace.

Table 1: 4-tile configuration results

Arch.	App.	Large Cfg.: 4 tiles, each tile has 4x4PEs			
		ops	cycles	Avg. IPC	Perf. Gain
ADRES	idct_row(8x8)	-	-	27.7	-
FDR-CGRA	idct_row(8x8)	857	24	35.7	29%
ADRES	idct_col(8x8)	-	-	33.0	-
FDR-CGRA	idct_col(8x8)	1185	33	35.9	9%
FDR-CGRA	interpolate8x8_avg4_c	1193	40	29.8	-
FDR-CGRA	interpolate8x8_halfpel_hv_c	1295	38	34.1	-
FDR-CGRA	sad16_c(16x16)	3441	106	32.5	-

Table 2: Single-tile configuration results

Arch.	App.	Small Cfg.: 1 tile, 4x4PEs each tile			
		ops	cycles	Avg. IPC	Perf. Gain
RCP	idct(row+col)	-	-	9.2	-
FDR-CGRA	idct(row+col)	2042	184	11.1	21%
FDR-CGRA	interpolate8x8_avg4_c	1193	136	8.8	-
FDR-CGRA	interpolate8x8_halfpel_hv_c	1295	135	9.6	-
FDR-CGRA	sad16_c(16x16)	3441	339	10.2	-
ADRES	get_blocks (64 PEs)	-	-	29.9(64 PEs)	
FDR-CGRA	get_block(H)	340	38	8.9	-
FDR-CGRA	get_block(V)	296	37	8.0	-
FDR-CGRA	get_block(V+H)	899	93	9.7	-
FDR-CGRA	get_block(H+V)	900	95	9.5	-
FDR-CGRA	Adjusted Avg. (4 tiles)	-	-	36.1(4 tiles)	21%

As can be seen in Tables 1 and 2, our approach achieves average *Instructions-Per-Cycle* (IPC) of 35.7 and 35.9 for idct_row and idct_col, respectively, on a 4-tile 4×4 FDR-CGRA, outperforming 8×8-FU ADRES architecture reported in [1] by 29% and 9% respectively. Our result of IDCT scheduled on a single tile 4×4 FDR-GCRA outperforms the 16-issue RCP's IPC value of 9.2 by 21%. The average IPC for "get_block" functions for H.264 on a single tile is about 9, and on 4 tiles it is more than 36, which is also better than the "in-house" optimized results (avg. IPC = 29.9) in [8].

References

[1] B. Mei et al., "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study", in *Proceedings of DATE*, p. 21224. 2004

[2] O. Colavin, et al., "A scalable wide-issue clustered VLIW with a reconfigurable interconnect," in *Proceedings of CASES*, pp. 148–158. 2003.

[3] M. B. Taylor et al., "Evaluation of the RAW microprocessor: an exposed-wire-delay architecture for ILP and streams," in *Proceedings of ISCA*, pp. 2–13 2004.

[4] G. Dimitroulakos, et al., "Design space exploration of an optimized compiler approach for a generic reconfigurable array architecture," *J. Supercomputing*, vol. 40, issue 2, pp. 127–157, 2007.

[5] S. Sait, et al, *VLSI Physical Design Automation: Theory and Practice.* Hackensack, NJ: World Scientific Publishing, 1999.

[6] C. Lattner, "Introduction to the LLVM Compiler Infrastructure," presented at the 2006 Itanium Conference and Expo, San Jose, California, April 2006.

[7] T. J. Todman, et al., "Reconfigurable computing: architectures and design methods," *IEE Proc. Comp. Digit. Tech.*, Mar. 2005.

[8] B. Mei et al., "Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture," in *Proceedings of FPL* , pp. 622–625, 2005