Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 163542, 17 pages
doi:10.1155/2012/163542

Research Article

A Coarse-Grained Reconfigurable Architecture with Compilation for High Performance

Lu Wan,[1] Chen Dong,[2] and Deming Chen[1]

[1] *ECE Illinois, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2918, USA*
[2] *Magma Design Automation, Inc., San Jose, CA 95110, USA*

Correspondence should be addressed to Deming Chen, dchen@illinois.edu

Received 5 October 2011; Revised 5 January 2012; Accepted 9 January 2012

Academic Editor: Kentaro Sano

We propose a *fast data relay* (FDR) mechanism to enhance existing CGRA (*coarse-grained reconfigurable architecture*). FDR can not only provide multicycle data transmission in concurrent with computations but also convert resource-demanding inter-processing-element global data accesses into local data accesses to avoid communication congestion. We also propose the supporting compiler techniques that can efficiently utilize the FDR feature to achieve higher performance for a variety of applications. 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 the new compiler deliver up to 29% and 21% higher performance than ADRES and RCP, respectively.

1. Introduction and Related Work

Much research has been done to evaluate the performance, power, and cost of reconfigurable architectures [1, 2]. Some use the standard commercial FPGAs, while others contain processors coupled with reconfigurable coprocessors (e.g., GARP [3], Chimaera [4]). Meanwhile, *coarse-grained reconfigurable architecture* (CGRA) has attracted a lot of attention from the research community [5]. CGRAs utilize an array of pre-defined *processing elements* (*PEs*) to provide computational power. Because the PEs are capable of doing byte or word-level computations efficiently, CGRAs can provide higher performance for data intensive applications, such as video and signal processing applications. In addition, CGRAs are coarse grained so they have smaller communication and configuration overhead costs compared to fine grained field programmable gate arrays (FPGAs).

Based on how PEs are organized in a CGRA, the existing CGRAs can be generally classified into linear array architecture and mesh-based architecture. In linear array architecture, PEs are organized in one or several linear arrays. Representative works in this category are RaPiD [6] and PipeRench [7]. RaPiD can speed up highly regular, computational intensive applications by deep pipelining the application on a chain of RaPiD cells. PipeRench provides speedup for pipelined application by utilizing PEs to form reconfigurable pipeline stages that are then interconnected with a crossbar.

The linear array organization is highly efficient when the computations can be linearly pipelined. With the emergence of many 2D video applications, the linear array organization becomes less flexible and inefficient to support block-based applications [8]. Therefore, a number of mesh-based CGRAs are proposed. Representative works in this category include KressArray [9], MATRIX [10], and MorphoSys [8].

The KressArray was one of the first works that utilized 2D mesh connection in CGRA. KressArray uses *reconfigurable data path units* (rDPUs) as basic computation cell. These rDPUs are connected through local *nearest neighbor (NN)* links and global interconnection [9]. To accommodate high volume of communications between function units, the MATRIX [10] proposed a 3-level network connecting its *basic function unit* (BFU) including NN connection, length-4 bypassing connections and global lines. Finally, Morphosys [8] used an 8×8 PE array, divided into four 4×4 quadrants, as the coprocessor for a RISC host CPU. Each PE is directly connected with any PE in the same row/column within the same quadrant and its NN regardless of the quadrants. Morphosys also exploits a set of buses called *express lanes* to link PEs in different quadrants.

Two interesting CGRAs are recently proposed: ADRES [11] and RCP [12]. ADRES belongs to mesh-based architecture. It utilizes Morphosys' communication mechanism and resolves resource conflict at compile time using modulo scheduling. RCP belongs to linear array architecture. It proposes an architecture with ring-shaped buses and connection boxes. One unique feature of RCP is that it supports concurrent computation and communication [12]. However, its hardware and software only support this feature in one dimension, which may limit RCP performance. Another representative architectural work is RAW [13], which pioneered the concept of concurrent computation and communication using MIPS processor array. However, due to the complexity of MIPS processor as a PE, RAW may not be an efficient platform to accelerate kernel code used in embedded applications. On the compiler side, a recent work is SPR [14], which presents an architecture-adaptive mapping tool to compile kernel code for CGRA. Their compiler targets an FPGA-like CGRA, and it supports architectures with a mix of dynamically multiplexed and statically configurable interconnects.

Some recent studies [15–18] show that further exploiting instruction level parallelism (ILP) out of ADRES architecture can cause hardware and software (compiler) inefficiency. This is mainly because ADRES utilizes heavily ported global register file and multi-degree point-to-point connections [18]. In particular, to implement communication pattern that broadcasts shared data on ADRES is challenging [15]. Using integer linear programming to find better code schedule may incur prohibitive long runtime [16]. Given the difficulty of further pushing for ILP performance, [17] turned to exploit thread-level parallelism in MT-ADRES.

In this work, to exploit ILP performance further, we propose a new mechanism, named *fast data relay* (*FDR*), in CGRA. This proposed FDR architecture replaces the heavily ported global register file with distributed simple register files (Section 2.2). It also replaces the point-to-point connections with simple wire-based channels (Section 2.3). The main features of FDR include (1) fast data links between PEs, (2) data relay where communication can be done in multiple cycles as a background operation without disturbing computations, and (3) source operand that can have multiple copies with longer lifetimes in different PEs so that a dependent PE can find a local copy in its vicinity. We name the proposed new architecture *FDR-CGRA*. These FDR mechanisms are generic and could also be incorporated into ADRES architecture to enhance the ILP performance. To utilize FDR efficiently, we propose new compiler techniques to map kernel code onto FDR-CGRA leveraging its unique hardware features.

In the following, we provide the motivation and the architecture support of FDR-CGRA in Section 2. In Section 3, we introduce our CAD-based compiler support for the efficient use of FDR. The experimental results will be discussed in Section 4, and we conclude this paper in Section 5.

2. Motivation and Architectural Support for *FDR*

With the shrinking of CMOS technology, more and more devices can be integrated into a single die. The granularity of CGRA has also gradually been raised from 4 bit to 16 bit or even 32 bit. The ADRES uses 32 bit ALU as its basic computation unit and is one of the representative modern CGRA architectures for mesh-based designs. It couples a VLIW processor with an array of PEs. Each PE consists of a predicated ALU, local register file, and I/O multiplexers. The functionality of each ALU can be configured dynamically through the *context SRAM* (CS). A unified *global register file* (GRF) is used to provide data exchange among PEs and the host VLIW processor. The interconnection of ADRES exploits the communication mechanism similar to Morphosys. Its compiler uses modulo scheduling algorithm to resolve resource conflict at compile time.

The research in [18] extensively studied how the configuration of register files impacts the performance of ADRES. It showed that the unified GRF is a severe performance bottleneck. In ADRES, each PE connected to the GRF has dedicated read and write ports to the GRF. In a 64-PE ADRES baseline configuration organized as 8 rows and 8 columns, eight PEs on the top row are connected to the GRF. The study in [18] showed that even this baseline configuration needs a highly complicated 8-read and 8-write GRF with significant area penalty. And at the same time, allowing only 8 simultaneous reads and 8 simultaneous writes to this GRF seriously limited the performance (IPC < 14) of 64-PE ADRES. It is also shown in [18] that to achieve an IPC above 23 on the original 64-PE ADRES architecture, the global register file needs to support >40 sets of read and write ports, which is, if not impossible, very challenging and costly for physical implementation. Also, their study pointed out that increasing the number of registers in GRF and each PE local register file can only contribute to performance marginally. Due to this limitation, the performance of ADRES can barely take advantage of the growing budget given by ever-shrinking modern CMOS technology.

RCP is a representative architecture for linear array-based design. But linear array architecture may not be able to accommodate block-based applications as well as mesh-based designs. To study this, we will also compare our results with RCP. In the rest of this section, we propose several architectural enhancements for the mesh-based architecture to improve the performance. Then, in the section afterwards, we will propose several compiler techniques for better utilizing these new hardware features. As will be shown in the experimental result section, these hardware changes as well as the new compiler techniques can exploit higher instruction level parallelism (ILP) comparing to ADRES and RCP.

The architectural changes are summarized into a single term named fast data relay (FDR), which includes three main features: *bypassing registers*, *wire-based companion channels*, and the use of *non-transient copies*. In the following subsections, we will first introduce the overall architecture of FDR-CGRA. Then, each of the aforementioned three features will be explained in detail, respectively.

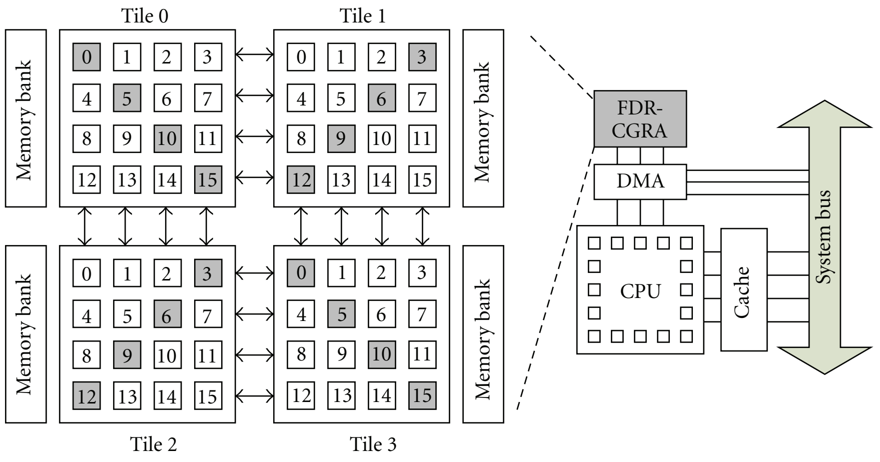


Figure 1: FDR-CGRA system overview.

2.1. Architectural Overview. Similar to existing reconfigurable computing platforms surveyed in [2], we assume that a host CPU initializes the kernel computation on our FDR-CGRA and is in charge of system-level dataflow via DMA. We assume filling data via DMA can be done in parallel with CGRA computations using double buffers (Ping-Pong buffers). 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 2.2) organized in tiles, and a wire-based communication layer (Section 2.3) 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 (Figure 1).

To exploit more instruction level parallelism, it is desirable to exchange data for operations on non-critical paths without intervening with critical computations. Fast data relay is such a mechanism. It is capable of routing data from one PE to another in multiple cycles in parallel with computations by utilizing the bypassing registers (Section 2.2) and companion channels (Section 2.3). It is an enhancement over ADRES point-to-point communication. The philosophy of concurrent computation and communication was exploited in the RAW microprocessor [13]. However, instead of using switches that route data dynamically, FDR-CGRA uses wire-based channels. Communication within a tile is faster in FDR-CGRA than in RAW. The detailed analysis can be found in Section 2.3.

2.2. Processing Element and Bypassing Registers. As shown in Figure 2, each PE has a dedicated computation data path (left) similar to the PE used in ADRES and a communication bypassing path (right) that can be used for FDR. This dedication is essential to enable simultaneous computation and communication. Similar to ADRES, each 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, and B) on the top of the PE and output multiplexers at the bottom (3 and 4).

One key hardware feature of FDR-CGRA is the use of distributed bypassing register file (marked as BR in Figure 2): in addition to the normal local register file (marked as LR), which is used to store intermediate data for ALU, our PE includes a bypassing register file, through which multicycle data communication can be carried out. A leading PE can deposit a value in the bypassing register of a leading PE and a trailing PE can fetch it later on without disturbing computations on the leading PE. Such multicycle data communication is an effective way to reduce communication congestion, because it can detour communication traffic for non-critical operations to uncongested area. Note that the way our compiler utilizes the bypassing registers is different from the way the PipeRench compiler utilized its passing registers. In PipeRench, the passing registers were solely used to form virtual connections when a connection needs to span multiple stripes. And due to the greedy nature of the PipeRench compiler [19], whenever a path is unroutable due to resource conflict, a NOOP is inserted as the solution. As we will point out in Section 3.2 and in the experimental results, our compiler will use the bypassing registers as a powerful resource to solve routing congestions and exploit more ILP from application.

The bypassing register file associated with each PE provides a way for information exchange among PEs without incurring the same design complexity as in ADRES. As pointed out in [20, 21], central register file with a large number of read/write ports involves considerable complexity from multiplexing and bypassing logic, resulting in area penalty and timing closure challenges. The authors of [20] advocated to use distributed register files for media processing instead of using a heavily ported monolithic central

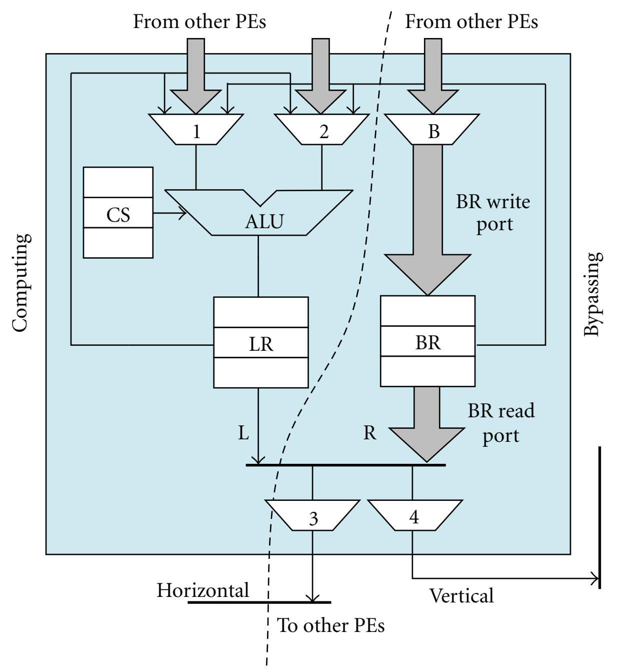


Figure 2: Architectural support for fast data relay at the PE level.

register file. Following this approach, in our FDR-CGRA, the distributed bypassing registers replace the unified GRF used in ADRES and its descendants [11, 15, 22]. To further simplify the register file design, we restrict the number of simultaneous accesses to the bypassing register file to 2 reads and 2 writes (2R2W). This constraint is explicitly modeled, and our tool can honor it during operation scheduling, mapping, and routing. By distributing the unified data storage in ADRES into small pieces as bypassing registers, we can exploit more ILP out of applications and gain larger programming flexibility over the original ADRES. However, at the same time, they pose some challenges for our compiler. This will be discussed in Section 3.

2.3. Wire-Based Companion Channels. In FDR-CGRA, inter-tile communication is simply through direct connections between PEs on the tile boundaries, as shown in Figure 1, while intratile communication is implemented with companion channels (Figure 3). Each PE owns two private companion channels: a vertical channel and a horizontal channel. Within a tile, the owner PE can send data via its private vertical (horizontal) channel to any PE in the same column (row) in a time-multiplexing way. Note that a vertical/horizontal channel can only be used by its owner for sending (not receiving) data. In this sense, these vertical (horizontal) channels are *unidirectional*. In Figure 3, for example, the companion channels owned by PE12 are PE12V and PE12H. All other PEs have similar sets of companion channels: there are four *unidirectional* vertical channels for each column of PEs and four *unidirectional* horizontal channels for each row of PEs. Note that vertical/horizontal channels span only across the PEs within the same tile for scalability reasons. Comparing to the 3-level interconnections used in Morphosys and ADRES where each PE has a set of dedicated links connecting to all PEs in the same row and column within a quadrant, our organization of companion channels is simpler because each PE only has one vertical and one horizontal channel and both are unidirectional. Initialized by the sender PE, scalar data can be deposited onto either horizontal or vertical channel through the output multiplexer. In 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.

Comparing with multipoint point-to-point connection used in other tile-based architecture [8, 11, 15], utilizing the proposed simple wire-based companion channels may have some benefits when it comes to area cost. Firstly, the regularity of these proposed companion channels makes them easier to be implemented on top of PEs in metal layers without complicated routing to save routing area. Secondly, for intra-tile communication, the proposed companion channels can actually save configuration bits. Multi-point point-to-point interconnection needs several bits to specify a destination PE out of several connected neighbors. Using the companion channels, to specify either the vertical or the horizontal channel requires only one configuration bit.

Each PE in our architecture is a simple programmable predicated ALU similar to the PE defined in ADRES with the goal of accelerating the program kernel. ADRES assumed that in a single hop (a single cycle) a signal can travel across 5 PEs [11]. RCP assumed that one hop can travel across 3 PEs because of the complexity of its crossbar-like *connection*

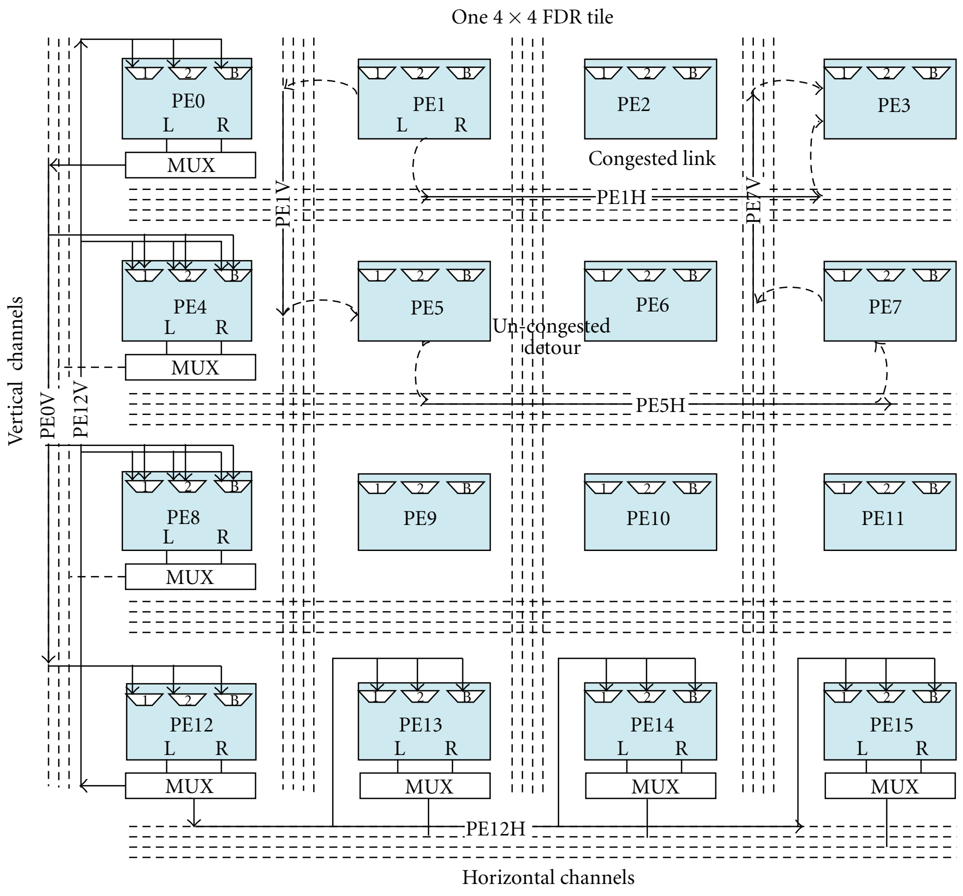


Figure 3: Companion channels and fast data relay.

box design [12]. RAW used complex MIPS core as PE, thus one hop can only travel across 1 PE [13]. Given the similarity of our enhanced PE to PE used in ADRES, 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 in 2 cycles (PE0 → PE12 → PE15) via PE0 vertical channel (PE0V) and PE12 horizontal channel (PE12H) as shown in Figure 3.

It should be noted that ADRES connects PEs with dedicated point-to-point links without the capability of multi-cycle routing and buffering as proposed in our FDR-CGRA. The bypassing registers and the companion channels enable us to do data communication in a more flexible way. The right upper corner of Figure 3 illustrates a possible scenario to do multi-cycle communication if data needs to be sent from PE1 to PE3 and the compiler finds the horizontal channel PE1H connecting from PE1 to PE3 is congested, then it can decide to use a multi-cycle detour through PE5 and PE7 by utilizing their bypassing registers and companion channels PE1V, PE5H, and PE7H to avoid the congestion. Comparing with RCP, FDR-CGRA supports 2D communication, rather than the 1D ring-shape communication in RCP. Furthermore, the companion channel also allows shared data to be sent concurrently to multiple PEs in the same row/column. These capabilities help alleviate the data route congestion problem experienced in other works (e.g., [15]).

Each communication link used in FDR involves three components: sender PE, companion channel, and receiver PE. We call such a communication path *inter-PE link*. It is desirable to reduce the total amount of inter-PE communication to reduce the latency and power. Next, we introduce the concept of non-transient copy to mitigate the inter-PE link usage in our solution.

2.4. Non-transient Copy. All PEs on the multi-cycle communication path except the sender 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 can possess a local copy of the origin data. Because our PEs contain bypassing register files, multiple local copies from different source PEs can coexist in the bypassing register file simultaneously. Furthermore, we can classify these local copies into two categories: *transient copy* and *non-transient copy*. Here we use "transient" to refer to a software property of a value hold in registers. We call a value a transient copy if the register holding the value will be recycled immediately after the value being fetched by the first trailing-dependent operation. In contrast, a non-transient

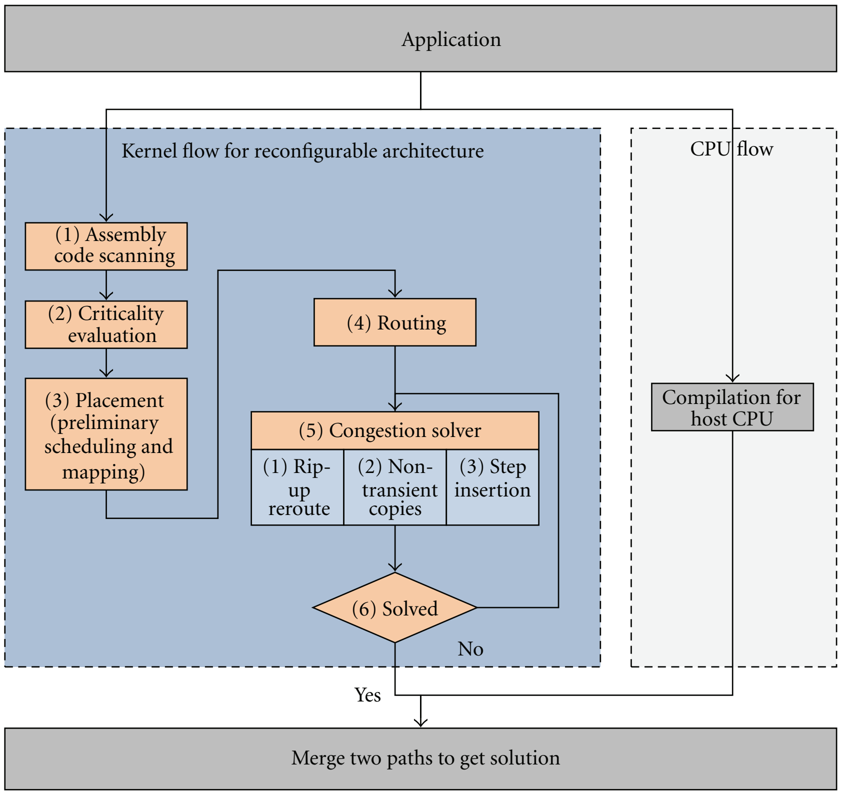

Application
Kernel flow for reconfigurable architecture
CPU flow
(1) Assembly code scanning
(2) Criticality evaluation
(3) Placement (preliminary scheduling and mapping)
(4) Routing
(5) Congestion solver
(1) Rip-up reroute
(2) Non-transient copies
(3) Step insertion
(6) Solved
No
Yes
Compilation for host CPU
Merge two paths to get solution

Figure 4: Overall flow.

copy will stay in the register until no more operations depend on it. Note that physically there is no difference between transient copy and non-transient copy since they both reside in the bypassing register file. Conceptually, the compiler treats them differently at compile time according to how they will be utilized. Since ADRES and RCP do not have bypassing register files to hold multiple copies, they need to recycle registers as soon as possible.

As will be shown in our experimental results, using non-transient copies can significantly benefit applications where some source data need to be reused by multiple dependent operations. Because it in effect distributes a source operand to several PEs and keeps them alive as non-transient copies, a dependent operation can probably find a local copy in its vicinity, instead of getting it from the origin PE. This can decrease the number of inter-PE links used.

From now on, we call those linking PEs that can provide local copies *identical start points*. Whether to keep the scalar data in linking PEs bypassing register files as a transient copy or a non-transient copy is decided at the routing stage (Section 3.2).

3. Supporting Compiler Techniques

Because FDR-CGRA is intended to accelerate the kernel code of an application, the overall design flow shown in Figure 4 can branch into two separate paths. On the right, the non-kernel code is compiled in a traditional manner targeting the host CPU. On the left is the flow to map kernel code operations onto the FDR-CGRA, which is the focus of our study.

First, we choose *low level virtual machine* (LLVM) [23] to compile the computation-intensive kernel code in C into assembly code. LLVM has its own architecture-independent virtual instruction set, and it can extensively rename registers during compilation, which is essential to avoid hitting the unnecessary IPC wall due to the limited number of registers that can be used by a compiler targeting specific processor architecture [24]. The assembly code of the kernel function generated by LLVM with the optimization option "–O3" is then scanned by a parser to identify data dependencies, and a directed acyclic graph (DAG) representing the data dependency among operations in the kernel code is constructed in step 1 of Figure 4. Step 2 will evaluate the criticality of each node in the DAG. This will be discussed in detail in Section 3.1.

Later on, steps 3–6 are used to find a valid operation mapping honoring resource constraints. Step 3 provides a preliminary placement for operations. Steps 4 and 5 refine it by doing routing and solving congestions. The problem of mapping kernel code onto FDR-CGRA can be formulated as follows: given a DAG representing the data dependency among operations, find a valid scheduling, mapping, and data communication scheme 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 Figure 5 by the following steps: (1) line up all PEs in the two-dimensional computation grid along the x

axis. The PEs on the grid are marked as PE(m, n), where m is the PE ID and n is this PE schedule step. (2) Duplicate the PE row L times along the time axis (the y axis) to form a routing region, where L is achieved through a list scheduling algorithm honoring the resource constraint. (3) Add edges between any nearby PE rows to reflect the available connectivity among PEs according to the connectivity specification in the architecture, that is, companion channels specified in Section 2.3. Figure 5(b) shows a sample routing region for a small group of PEs specified in Figure 5(a). The solid lines indicate inter-PE connections and the dash lines indicate intra-PE communication. Such a routing region incorporates all the feasible connections between PEs along the scheduling steps and provides a working space to carry out the compilation tasks.

Figure 5(c) is a DAG representing data dependency among operations. In this DAG, (1) solid edges represent single-cycle data dependency and (2) the dashed edge represents multi-cycle dependency, which requires FDR. Our compilation process consists of two phases: *placement frontend* and *routing backend*. The placement frontend produces an operation mapping solution in the routing region and leaves the detailed resource conflict problems to be resolved in the routing backend. This is similar to the physical design stages where a placer determines the location of cells and the routing determines the detailed wiring of the cells. In our case, each cell would be an operation. Figure 5(d) shows a placement and routing result for the DAG in Figure 5(c). In cycle 1, nodes 1 to 4 are mapped to PEs 0 to 3, respectively. In cycle 2, node 5 is mapped to PE 1 and node 6 to PE 3. Note that the output of node 2 is kept in PE 1 bypassing register file, which is then used by node 8 in cycle 3.

We would like to emphasize that explicitly using a dedicated routing phase to solve congestion can enable us to achieve higher ILP performance than modulo scheduling used in CGRA compilers from [11, 12, 14]. When resource conflict due to the competition for the same resource by consecutive iterations in a loop cannot be resolved, modulo scheduling algorithm resorts to increasing the *iteration interval* (II), which has negative impact on ILP performance. Meanwhile, although modulo scheduling is commonly used to optimize loops with arbitrary large number of iterations, the performance could suffer from the unsaturation 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. In contrast, our two-phase procedure can virtually discard the II constraints by unrolling the whole loop and explicitly resolving resource conflicts during the backend data routing stage. Considering that the routing algorithm we borrowed from the physical design automation field can typically handle hundreds of thousands of instances, this dedicated routing phase enables us to work on the fully unrolled kernel loops efficiently.

3.1. Placement Frontend for FDR

3.1. Placement Frontend for FDR . A criticality evaluation needs to be done first for all nodes in the DAG to guide placement. Note that we will use the term node and operation interchangeably from now on. The node criticality is evaluated as *slack* defined as

$$slack(op) = ALAP\_level(op) - ASAP\_level(op).$$

ALAP_level (*ASAP_level*) is the as-late-as-possible scheduling level (as-soon-as-possible scheduling level) for an operation computed by the ALAP (ASAP) scheduling algorithm [25]. If the slack is zero, there is no margin to delay the execution of the operation. If the slack is a positive value, the operation can be delayed accordingly without hurting the overall schedule length.

The placement frontend carries out operation scheduling and operation-to-PE mapping. We perform a list scheduling [25] guided by operation criticality such that critical operations will be placed on PEs that are directly connected with companion channels and noncritical operations will be placed on PEs without direct companion connecting them.

Algorithm 1 outlines the algorithm. First of all, every unprocessed node (operation) whose dependency is satisfied (i.e., its predecessor nodes, including PI nodes, are already scheduled and mapped) in the DAG is collected into *ReadyNodeSet*. Given the *ReadyNodeSet*, *FreePESet* (available PEs) and the current schedule step S as inputs, the algorithm maps the nodes from *ReadyNodeSet* to PEs in *FreePESet* and output a *MappedNodeSet*. Two constraints are considered here (lines 2–12 of Algorithm 1).

3.1.1. Functionality Constraint. This is to make sure that the candidate PE has the ability to perform the required computations because the PEs could be heterogeneous. For example, not all ALUs can perform memory load/store operations as explained in Section 2.1. We use *CapablePESet(n)* to identify the set of PEs that are capable of performing the operation n (line 3).

3.1.2. Routability Constraint. Although detailed routing scheme is determined at routing backend, a hint of routability is still used to help Algorithm 1 to determine mapping of nodes by calling *RoutableSet(p, l)* (line 8-9). Recall that FDR may need multiple steps to send data from a source PE to a destination PE. Given a PE p, the l-step reachable set of p is defined as all the PEs which can be reached within at most l hops starting from p. Then the candidate PEs are filtered out by taking the intersection of candidate sets generated according to functionality constraints, routability constraints, and availability constraints (line 10).

As an example, assume operation node b has two source operands: one is on PE1, and other is on PE2. Assume we have

$$CapablePESet(b) = \{\text{PE0 PE2 PE4 PE8}\},$$
$$RoutableSet(PE1, l) = \{\text{PE4 PE5 PE7}\},$$
$$RoutableSet(PE2, l) = \{\text{PE3 PE4 PE9}\}.$$

The intersection of the above three sets is PE4, indicating that PE4 is the only mapping candidate for node b. If PE4 turns

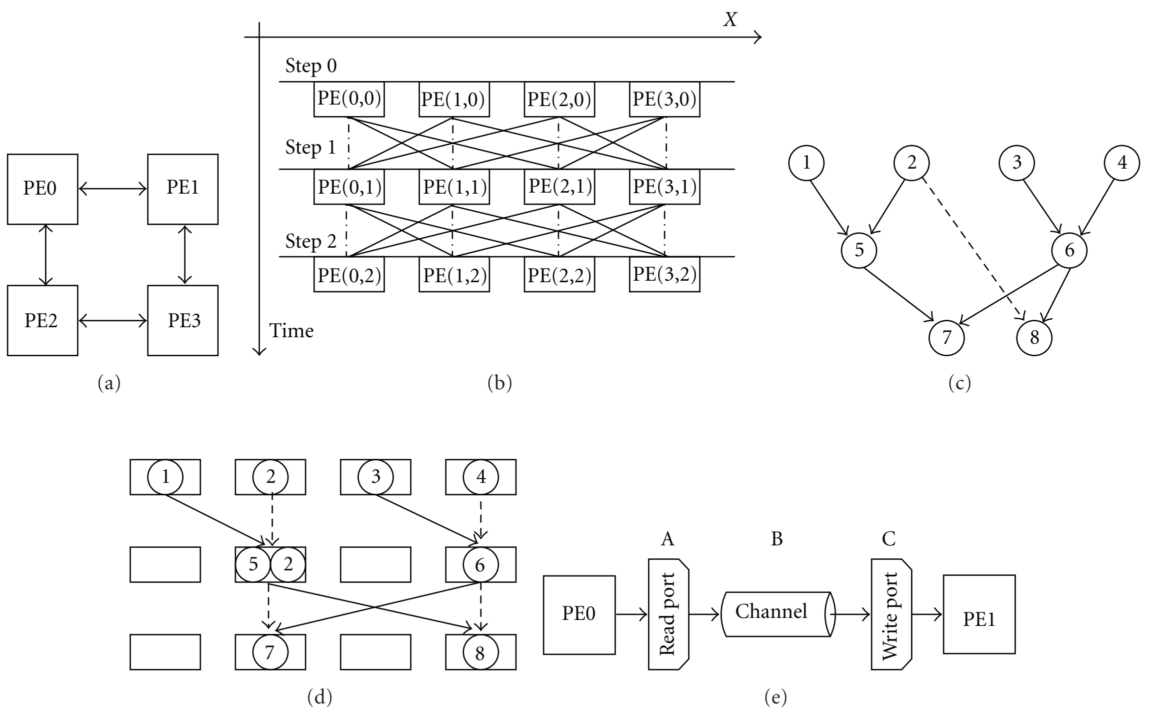


Figure 5: (a) A 2 × 2 PE array, (b) the routing region, (c) the DAG, (d) the mapping and routing solution, and (e) constraint model

```
Input: ReadyNodeSet, FreePESet, current schedule step S
Output: MappedNodeSet
(1) for each $PE_i$ in FreePESet {CandidateNodeSet($PE_i$) = {empty};}
(2) for each node n in ReadyNodeSet {
(3)    CapablePESet(n) = all PE capable of the function of n;
(4)    S_src1 = the time when n's operand 1 is produced;
(5)    L_src1 = the PE where n's operand 1 is stored;
(6)    S_src2 = the time when n's operand 2 is produced;
(7)    L_src2 = the PE where n's operand 2 is stored;
       //all PEs which can be reached within
       //(S-S_src1) cycles from PE L_src1;
(8)    RoutableSet1 = RoutableSet(L_src1, (S-S_src1));
       //all PEs which can be reached within
       //(S-S_src2) cycles from PE L_src2;
(9)    RoutableSet2 = RoutableSet(L_src2, (S-S_src2));
(10)     CandidatePESet = RoutableSet1 $\cap$RoutableSet2
                  $\cap$FreePESet $\cap$CapablePESet(n);
(11)     for each $PE_i$ in CandidatePESet{Add n to CandidateNodeSet ($PE_i$);}
(12)   }
(13)   MappedNodeSet = {empty};
(14)   for each $PE_i$ in CandidatePESet{
(15)     Sort all nodes in CandidateNodeSet ($PE_i$) according to criticality;
(16)     M = most critical from CandidateNodeSet and
              M $\notin$MappedNodeSet;
(17)     Place M on $PE_i$ at schedule step S;
(18)     Insert M into MappedNodeSet;
(19)     Remove M from ReadyNodeSet;
(20) }
```

Algorithm 1: Placement frontend (simultaneous scheduling and mapping).

out to be already occupied (PE4 not free), node b cannot be mapped in this schedule step and will fall to later schedule steps.

If there are more than one candidate nodes to map into a PE as indicated in *CandidateNodeSet*, the node with higher criticality will have a privilege over those with lower criticality for mapping (line 13–19). Those successfully mapped nodes will be moved to the *MappedNodeSet* (line 18), while the unmapped nodes will still remain in the *ReadyNodeSet* to be processed in scheduling steps thereafter. This procedure will continue until all nodes are processed.

3.2. Routing Backend for FDR. The routing backend is used to solve resource conflict in detail. We use an algorithm inspired by global maze routing [26] to find inter-PE communication schemes that can efficiently utilize bypassing registers introduced in Section 2.2. Our algorithm also determines which operands stay in bypassing registers as non-transient copies and which are not.

Each operand used in the DAG is identified with its own ID and its associated location in the routing region in the format of PE$[m, n]$, where m stands for the ID of PE that hosts the operand and n stands for the schedule step in which the operand is ready for R/W (read/write). The inputs to the routing procedure are a list of routing requirements represented as $(src \rightarrow dest)$ pairs, where *src* and *dest* are in the above format. Figure 6 shows an example. Assume three operations op1, op2, and op3 all need an operand M from PE $[1, 1]$ as indicated in Figure 6(a) with 3 routing requirements:

$$\begin{aligned}&\text{op1: } M, \text{PE}[1,1] \rightarrow \text{PE}[2,4],\\ &\text{op2: } M, \text{PE}[1,1] \rightarrow \text{PE}[0,4],\\ &\text{op3: } M, \text{PE}[1,1] \rightarrow \text{PE}[3,5].\end{aligned}$$

3.2.1. Solve Routing Congestion with Rip-Up and Reroute. An initial routing solution usually has congestion. We improve the quality of routing solutions iteratively. A commonly used way to solve congestion is by rip-up and reroute [26] as shown in Algorithm 2: (1) pick a routed path (line 4); (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 (line 8); (3) based on the existing congestion information of the routing region, find an uncongested detour and reroute the path (line 11–14). These steps are iterated for all paths until the congestion is eliminated or it stops when no improvement can be achieved. After rip-up and reroute, a routing solution for the previous problem is shown in Figure 6(b), in which four inter-PE links (solid arrows) are used.

3.2.2. Detail Route for Each Inter-PE Communication Path. To detail route for each communication path, a maze routing procedure "route_path" in line 11 of Algorithm 2 is called to find routing solution for each requirement entry. Note that a set of special data structures used for CGRA is used to provide routing guidance for "route_path." First, we apply *channel constraints*, which are channel capacities on vertical and horizontal companion channels and apply *port constraint* which is an upper bound on the number of total simultaneous accesses to the read/write ports of bypassing register files in PEs to reflect the bandwidth constraint of a physical register file (Figure 2). Second, each inter-PE communication link is modeled as a cost function of three parts (Figure 5(e)): read port, channel, and write port. Whenever an inter-PE link is used, the capacity of the affected channel and ports will decrease and the cost of using this link will increase proportional to the congestion on it. Third, each PE has a PE score recording the cost of the cheapest path (with least congestion) from the source PE.

At the beginning of "route_path," a queue contains only the source PE. Starting from the source PE by popping it up from the queue, multiple-directional scores are calculated for its trailing PEs, that is, directly connected neighboring PEs. PE scores for those trailing PEs may be updated to record the changes for the cheapest paths leading to them. Any PE whose PE score is updated will be pushed into the queue for further exploration. The process is quite similar to wave front propagation, and the router stops when the queue is empty. After wavefront propagation is done by "route_path," a "backtrace" procedure (line 13) starts from the end point and backtraces along the cheapest direction at each backtrace step in the routing region "cost_grid" until reaching the start point. This backtraced path "new_p" is reported as the cheapest path.

3.2.3. Utilize Non-Transient Copies. What differentiate our algorithm from ordinary routing algorithm is that we integrate the utilization of features of non-transient copy in the routing algorithm.

In the example of Figure 6(b), after the first route from M on PE$[1, 1]$ to op1 on PE$[2, 4]$ is established via the path PE$[1, 1]$ $\rightarrow$ PE$[3, 2]$ $\rightarrow$ PE$[2, 3]$ $\rightarrow$ op1. Both PE2 and PE3 have once possessed a copy of M during this procedure. In fact, the copy in PE3 can be reused by op3 later on at step 5, as long as the register holding value M in PE$[3, 2]$ will not be reclaimed immediately after PE$[2, 3]$ fetches it. This is achieved by marking the copy of M in PE$[3, 2]$ as a non-transient copy. In contrast, for the copy of M in PE$[2, 3]$, except op1 no other operation needs it. Therefore register holding M in PE2 can be marked as transient and reclaimed immediately after op1 consumes it to save register usage. To decide whether a copy should be transient or non-transient, a special data structure "regDuplicatorLst" is used in Algorithm 2 to help making the decision. "regDuplicatorLst" is implemented as a lookup table storing the identical start points for each origin source operand. The index to the table is the origin source operand, and the contents pointed by the index are all identical start points organized as a list. This lookup table is updated (line 15) every time after the cheapest path is found by the backtrace procedure by inserting all new identical start points into the entry of their associated origin source operands. An example of a "regDuplicatorLst" entry is shown below. After the first route to op1 in Figure 6(b) is found by "backtrace," the origin source M may be found in three different places:

$$M\text{: PE}[1,1], \text{PE}[3,2], \text{PE}[2,3].$$

```
Input: &routingPathLst // routingPathLst stores routing requirements
Output: a new less congested routing solution stored in routingPathLst
for (i = 0; i < iMax; i++) {
regDuplicatorLst.clear(); // clear regDuplicatorLst before new iteration
newRoutingPathLst.clear();
for each path p in routingPathLst {
  src = p.origin;    // get start_point of p
  dest = p.destination;    // get end_point of p
  //step 1: ripup
  ripup_path(p);        // ripup p by release all resources used by p
  end_step = getReadyStep(dest);
    //step 2: reroute
    cost_grid = route_path(src, end_step, &regDuplicatorLst);
    //step 3: backtrace
    new_p = backtrace(dest, &src, cost_grid);
    add(new_p, newRoutingPathLst);
    update(regDuplicatorLst, new_p);
}
    routingPathLst = newRoutingPathLst;
}
```

Algorithm 2: ripup_reroute.

As we pointed out before, an identical start point can serve as an alternative source for the same operand. To take this into consideration, procedure "route_path" of Algorithm 2 is modified accordingly to push all identical start points for the origin source into the queue at the beginning of the routing. As a result, the wavefront propagation can actually start from multiple sources. But only one source that produces the cheapest route found by "backtrace" will be used as actual start point. If this actual start point is different from the origin source, it means this data relay route actually reuses an operand that is deposited by another route before. To allow this reusing, this start point is marked as a non-transient copy.

In the simple example of Figure 6(c), by maintaining the identical start points in "regDuplicatorLst," the backtrace for op3 is able to find the cheapest route solution, which is to reuse the value M once stored in the bypassing register, for example, BR[2], of PE$[3, 2]$. Then BR[2] is declared as nontransient copy thus can be kept untainted in PE$[3, 2]$ for future reuse by op3 without establishing a physical inter-PE link from the origin of M in PE$[1, 1]$. With non-transient copies, the solution can make less use of inter-PE links, for example, in Figure 6(c) only three inter-PE links are used while four are used in Figure 6(b). Moreover, during the routing for op2, with the "regDuplicatorLst," Algorithm 2 will find it can get a copy of M from either PE$[1, 3]$ or PE$[2, 3]$ in Figure 6(c). With more choices, the router in Algorithm 2 can effectively select the least crowded communication scheme to avoid congestion.

Non-transient copies usually have longer life cycles than transient copies and serve as data origins for multiple trailing-dependent operations. Our compiler can trace the last access to a non-transient copy at the compile time and decide to reclaim this non-transient copy after this last usage.

3.2.4. Schedule Step Relaxation. In some cases, applying the above techniques still cannot resolve all the congestions, and *schedule step relaxation* (SSR) will then be invoked as the last resort. SSR utilizes a well-known fact that routing sparsely distributed nodes on a larger routing region will result in less congestion. SSR in effect increases the size of the routing region along the time axis by inserting extra steps into the most congested regions. Trading area for routability guarantees that congestion will be reduced to zero. As the last resort, a greedy algorithm is used to do SSR as follows: starting from the most congested step, insert an extra *relaxation step*, then redo rip-up and reroute on the extended routing region, and continue doing so until all congestions are solved. The side effect of SSR is the increased schedule length.

4. Experimental Results

Experiments are performed on an architecture configuration shown in Figure 1. The reconfigurable array consists of 4 tiles. Each tile has 16 PEs organized in a 4×4 mesh. The PEs in a single tile communicate with each other as specified in Section 2.3. The data relay ports on the bypassing register for read and write are bounded by 2, respectively, to reflect the physical register file bandwidth constraint. Similarly, load/store operations are limited to be performed only by PEs on the diagonal line to reflect memory bandwidth constraint. Other computational operations can be performed by any PEs. The latency of each operation is set to 1. LR has 16 entries. To explore the best performance that can be achieved using BR, the size of BR is not fixed. However, as the results shown in Section 4.3, a fairly small amount of BR is used, thus practical. The architecture in Figure 1 can be configured in two ways: (1) *large Cfg.*, all 64 PEs can be used to map

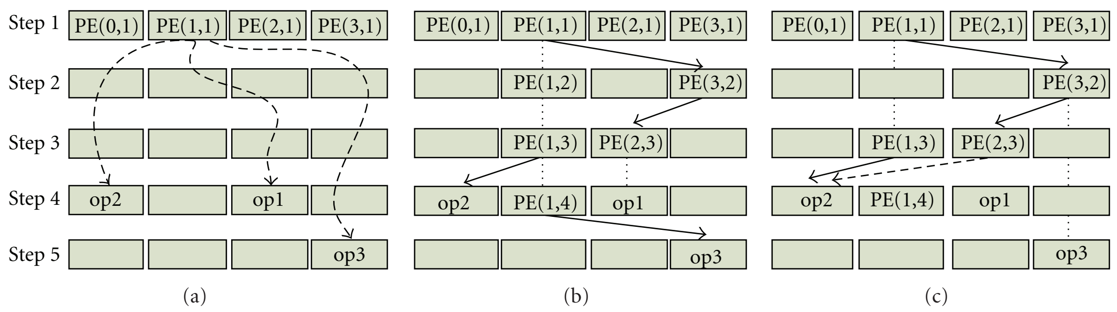


Figure 6: (a) Initial routing requirements, (b) data relay solution, and (c) solution of data relay with register duplication.

a benchmark program, (2) *small Cfg.*, a benchmark program is limited to be run on a single tile, but four independent programs can run concurrently on four tiles. Each Cfg. has its own merit to be explained in Section 4.2.

Benchmarks are extracted from two open source applications: xvid 1.1.3 and JM 7.5b. xvid [27] is an open source package consisting of various codecs for various video formats. JM v7.5b [28] is a reference implementation of the H.264 video codec. Five subroutines are extracted from the xvid package: idct_row, idct_col, interpolation8 × 8_avg4_c, interpolate8 × 8-_halfpel_*hv*_c, and SAD16_c. IDCT is a fixed-point inverse discrete cosine transformation working on rows and columns of the 8 × 8 pixel blocks. This is the same as that used in [11] and [12]. Interpolation8 × 8_avg4_c and interpolate8 × 8_halfpel_*hv*_c are subroutines to calculate pixel values for 8 × 8 blocks by interpolating pixels from neighbor image frames. SAD16_c is used in motion estimation to calculate the similarity of two 16 × 16 pixel blocks. Another four subroutines are extracted from the H.264 reference implementation JM v7.5b: getblock(V), getblock(H), getblock(HV), and getblock(VH). They calculate the pixel values for a 4 × 4 block by interpolating from neighbor image frames.

We should point out that the number of loop iterations in media applications is bounded by fine-grained partitioned data set, usually for better image quality. For instance, each H.264 loop typically operates on a 4 × 4 or 8 × 8 pixel block. This type of usage gives rise to the opportunity to exploit ILP within loops by fully unrolling, which fits perfectly to FDR-CGRA typical usage.

4.1. Execution Trace. Our compiler implemented in C++ can dump a placed and routed execution trace for the kernel code. Then a simulator implemented in PERL is used to validate the result. We use PERL because it supports text array very well, and text arrays are used to represent instruction, channel, and port configurations in our code. The simulator not only simulates the data flow on FDR-CGRA but also verifies on the fly for the satisfaction of three sets of resource constraints: channel constrains, port constraints and resource constraints. Table 4 shows an abbreviated execution trace for idct_row scheduled on an architecture depicted in Figure 1. To save space, only operations on one tile are shown and the other three tiles are omitted. 16 PEs in the tile are numbered from 0 to 15. Operations are represented as parenthesized letters. Round bracket ones are of one unrolled iteration (ite1), while square bracket ones are of another iteration (ite2). Operations from the same iteration can be scheduled across tiles when necessary. Both iterations start at the first cycle. In our approach, iterations compete for PE resource and also complement each other when the other is waiting for data: for example in Table 4, nine ops are from ite1 in cycle 5, while six ops are from ite2, while in cycle 6, five ops are from ite1 and ten ops are from ite2. In addition, we can see that the result respects the resource constraints by (1) only allowing PE0, 5, 10, 15 to perform load/store operations (L) due to memory bandwidth constraint (2) and only allowing at most 2 reads and 2 writes to each PE bypassing register file due to the port constraint (2R2W) we set. In Table 4, the “write (read) requests” column shows, for each PE, how many write (read) requests are served at each cycle. Each single digit from left to right in a row corresponds to the number of write (read) requests on PE0 to PE15, respectively. Each column of these digits reflects the write (read) requests for its corresponding PE over time. The “Communication Vol.,” where each * represents an inter-PE link usage, shows the amount of inter-PE link usage at each cycle. It is clear that computation and communication are indeed concurrently executed every cycle due to dedication of bypassing register, except cycle 1, 7, 10, 12, where extra relaxation steps are used to solve routing congestion.

4.2. Performance Analysis. Modulo scheduling-based code scheduling approaches [11, 15] have advantage to handle loops with a large number of iterations because they can reuse the scheduled code on PEs throughput the iterations. However, the inflexibility of modulo scheduling may limit its performance for a class of applications that have a special communication pattern. This type of applications, represented by video applications, usually “broadcast” shared data from one PE to multiple receiving PEs. As reported in [15], an edge centric modulo scheduling does not perform well for H.264 if an CGRA does not provide complex routing schemes including using diagonal channels.

The performance of benchmark programs on FDR-CGRA, as well as on ADRES and RCP, is shown in Tables 1 and 2. The benchmark programs are mapped on

Table 1: Large configuration results.

Arch.	App.	Large Cfg.: 4 tiles, each tile has 4 × 4 PEs				
		Ops	Cycles	Avg. IPC	Perf. gain	Efficiency
ADRES	idct_row(8 × 8)	—	—	27.7	—	43%
FDR-CGRA	idct_row(8 × 8)	857	24	35.7	29%	56%
ADRES	idct_col(8 × 8)	—	—	33.0	—	52%
FDR-CGRA	idct_col(8 × 8)	1185	33	35.9	9%	56%
FDR-CGRA	Interpolate 8 × 8_avg4_c	1193	40	29.8	—	47%
FDR-CGRA	Interpolate 8 × 8_halfpel_*hv*_c	1295	38	34.1	—	53%
FDR-CGRA	sad16_c(16 × 16)	3441	106	32.5	—	51%

Table 2: Small configuration results.

Arch.	App.	Small Cfg.: 1 tile, 4 × 4 PEs each tile				
		Ops	Cycles	Avg. IPC	Perf. Gain	Efficiency
RCP	idct(row+col)	—	—	9.2	—	57%
FDR-CGRA	idct(row+col)	2042	184	11.1	21%	69%
FDR-CGRA	interpolate8 × 8_avg4_c	1193	136	8.8	—	55%
FDR-CGRA	interpolate8 × 8_halfpel_*hv*_c	1295	135	9.6	—	60%
FDR-CGRA	sad16_c(16 × 16)	3441	339	10.2	—	63%
ADRES	get_blocks (64 PEs)	—	—	29.9(64 PEs)	—	47%
FDR-CGRA	get_block(H)	340	38	8.9	—	56%
FDR-CGRA	get_block(V)	296	37	8.0	—	50%
FDR-CGRA	get_block(V+H)	899	93	9.7	—	60%
FDR-CGRA	get_block(H+V)	900	95	9.5	—	59%
FDR-CGRA	Adjusted Avg. (4 tiles)	—	—	36.1(4 tiles)	21%	56%

both a large and a small Cfg's. In the large Cfg., our approach achieves an average IPC of 35.7/35.9 for idct_row/idct_col, outperforming 8 × 8-FU ADRES architecture reported in [11] by 29% and 9%, respectively. In the small Cfg., we compare our results with 16-issue RCP reported in [12]. Our result of IDCT scheduled on a single tile (16 PEs) outperforms the 16-issue RCP IPC = 9.2 by 21%. As to the get_blocks, we observed that some modes—get_block(H) and get_block(V)—have significant fewer ops than other benchmark programs. Spreading a small number of ops on a large number of PEs would artificially introduce many multi-hop routing requirements. Given that each get_block can indeed work independently of others, a more efficient way is to schedule each get_block on a single tile and allow four instances of different get_block's to run concurrently. With this 4x adjustment to the get_block's average IPC in Table 2, the final average IPC for get_block is larger than 36, which is better than the "in-house" optimized results (avg. IPC = 29.9 for ADRES) in [22]. Note that the IPC values for ADRES and RCP are directly quoted from their published results. No data is reported for the interpolation and SAD benchmarks from either ADRES or RCP.

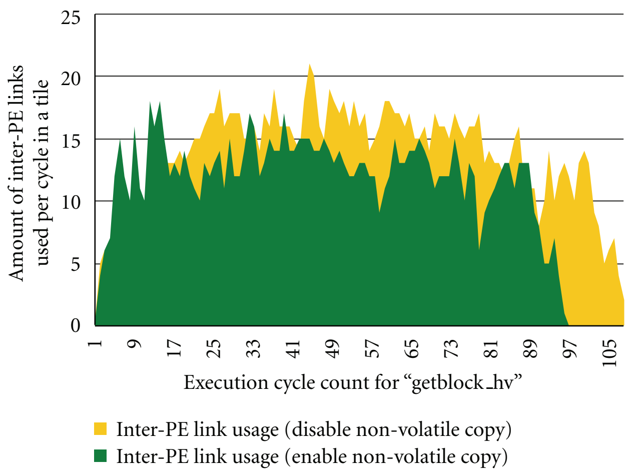
In the rightmost column of Tables 1 and 2, efficiency of a schedule is defined as the quotient of average IPC divided by total number of PEs. It can be observed that for the same application the large Cfg. has the smallest latency (under the *cycles* column) because it can use more resources, while the small Cfg. always has higher efficiency. This is because scattering operations on large Cfg. may introduce more operation "bubbles", where no useful work is actually done by a NOP operation for dependency or communication reasons. This fact suggests that the large Cfg. can be used when reducing execution latency is the sole goal, while the small Cfg. is suitable for applications where efficiency and performance are both important.

We should point out that one important reason that the proposed FDR-CGRA outperforms ADRES is the use of companion channels and non-transient copies. When necessary, the wire-based companion channel naturally allows "broadcast" of a datum to multiple PEs in the same row or column. At the same time, non-transient copies not only allow reusing of datum that needs to be shared by multiple PEs but also increase the path choices during routing.

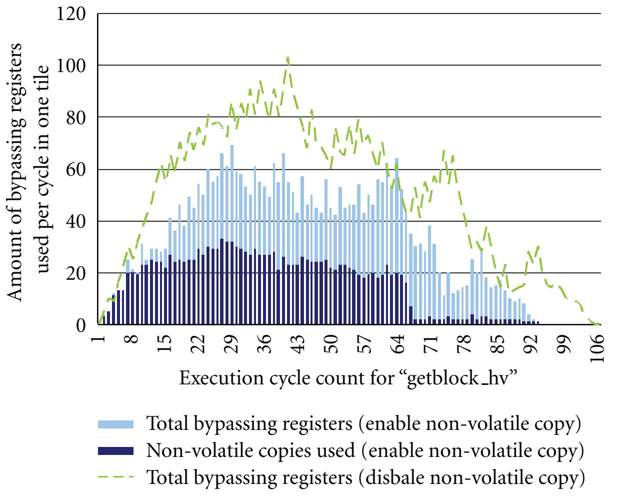
The comparison with RCP is also interesting because RCP also allows concurrent data routing and computations. However, it is limited to one dimension. The experimental results demonstrate that our proposed compiling technique effectively exploits the performance potential of FDR 2D concurrent data routing and computations.

Table 3: Bypassing register usage profile and effects of using non-transient copy on a single tile (16 PE) configuration.

	Bypassing register usage profile: per PE per cycle					Performance impact					
	Average		Peak		% Nontransient	Amount of inter-PE links			IPC		
	Disable	Enable	Disable	Enable	Enable	Disable	Enable	Delta	Disable	Enable	Delta
idct(row+col)	2.8	2.3	19	20	45%	2357	2245	−5%	10.9	11.1	2%
interpolate8 × 8_avg4_c	4.8	3.1	21	14	58%	1707	1333	−22%	6.4	8.8	38%
interpolate8 × 8_halfpel_*hv*_c	3.7	3.9	19	16	45%	1835	1621	−12%	8.3	9.6	16%
sad16_c(16 × 16)	1.0	0.8	6	5	54%	4752	4000	−16%	9.8	10.2	4%
get_block(horizontal)	1.0	0.9	6	5	22%	455	438	−4%	8.1	9.0	10%
get_block(vertical)	1.6	1.0	8	5	52%	513	400	−22%	5.7	8.0	41%
get_block(V+H)	4.2	2.2	18	10	45%	1469	1150	−22%	7.9	9.7	23%
get_block(H+V)	3.1	2.3	13	8	43%	1455	1148	−21%	8.4	9.5	13%
Average								−15%			18%



(a) Inter-PE link usage for each execution cycle with nontransient copy enabled and disabled



(b) Bypassing register usage for each execution cycle with nontransient copy usage breakdown

Figure 7: Performance impact of nontransient copies.

4.3. Effect of Non-Transient Copies. To test the effect of using non-transient copies, we rerun all the applications in Table 2 without using non-transient copies for a single tile configuration. The experimental results in Table 3 indicate that our algorithm produces solutions that only use an affordable amount of bypassing registers. On average, each PE only uses several bypassing registers at each cycle (column 2-3) and the peak bypassing register usage of a single PE ranges from 5 to 21 depending on applications. When non-transient copy usage is enabled, 22–58% of the total bypassing registers hold non-transient copies.

Table 3 also shows the performance impact of enabling non-transient copy. We can view its effects from three perspectives: (1) the primary goal of using it is to mitigate data relay congestion by allowing data reusing. With less congestion, the applications can finish sooner. (2) Distributing non-transient copies across a tile enables a PE to find a local copy in its vicinity and thus can reduce the usage of resource-and-power-consuming inter-PE links. (3) With less usage of inter-PE links, the total number of bypassing registers required can actually be reduced, although those non-transient copies do not recycle as fast as transient copies. These three effects are reflected in Figure 7 for application "getblock_hv." The usage for inter-PE links and usage for bypassing registers are plotted for every cycle. Obviously in Figure 7(a), non-transient copies enable the application to finish earlier and use less inter-PE links in total. It can be seen in Figure 7(b), with non-transient copy enabled, the proportion of non-transient copies to total bypassing registers is close to 1 in the first few cycles ranging from cycle 1 to 12 that correspond to loading input values from a reference image block. It means most of these loaded input values will be reused later on. Correspondingly, in Figure 7(a) the inter-PE link usage is also higher in the first a few cycles because non-transient copies need to be distributed across the tile as soon as possible for further reusing. But interestingly the total amount of bypassing registers used can still be kept lower with non-transient copy enabled. This is because that by reusing non-transient copies, the compiler does not need to allocate as many fresh

Table 4: Execution trace of IDCT (row).

	PE																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Communication Vol.	Write requests	Read requests	
Cycle 0:	(L)	—	—	—	—	[L]	—	—	—	—	—	—	—	—	—	—		0000000000000000	1000010000000000	
Cycle 1:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	**	0100100000000000	1100100000000000	
Cycle 2:	(P)	(P)	(P)	—	[P]	[P]	[P]	—	—	—	—	—	—	—	—	—	***	0011001000000000	0111111000000000	
Cycle 3:	(L)	—	—	—	—	(L)	—	—	—	—	(L)	—	—	—	—	(L)	******	0000011110100001	1000121100200002	"L": load
Cycle 4:	(C)	(C)	(C)	(C)	(C)	[L]	(C)	(C)	—	—	[L]	(C)	—	—	—	[L]	**********	0111111100110001	2222122110210002	"S": store
Cycle 5:	(A)	(A)	[C]	[C]	[C]	(M)	(M)	(M)	(M)	[C]	[L]	[C]	(M)	—	(M)	(H)	********************	2211221112111011	0111201002221001	"P": getptr
Cycle 6:	(M)	(M)	[A]	[M]	[M]	(A)	[M]	(A)	[C]	[A]	[M]	[M]	(H)	[C]	[C]	—	**************	0011111111111111	1101001110111111	"C": cast
Cycle 7:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	************	0121101111101001	0110110110100011	"M": mul
Cycle 8:	(B)	(B)	(B)	(A)	[A]	(M)	[M]	—	[M]	[M]	[H]	—	[H]	—	—	(B)	*********	1111101000110001	1122122111001001	"A": add
Cycle 9:	(B)	(B)	[B]	(A)	[B]	[M]	(A)	(B)	[A]	[B]	[A]	(A)	—	—	—	—	****************	2112111111020011	1101111111111001	"B": sub
Cycle 10:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	*************	1111111111110100	1011001111120111	"H": shl
Cycle 11:	(B)	(A)	(C)	(B)	[B]	[A]	(A)	(B)	(A)	[B]	(A)	[A]	—	—	—	—	*************	1001011111221001	1011110011111000	"R": shr
Cycle 12:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	**********	1010110101101101	1010211101110000	
Cycle 13:	(M)	(M)	(C)	(C)	[B]	[B]	(C)	[B]	(C)	(C)	[B]	[A]	[A]	[A]	—	[A]	**********	0001100100121111	0012110112110111	
Cycle 14:	(A)	(A)	(B)	[C]	[B]	[A]	[C]	[C]	(B)	[C]	(A)	[C]	[C]	—	—	(A)	**************	0111111111210002	0102001101122001	
Cycle 15:	(R)	(R)	(C)	(R)	[M]	[M]	(R)	—	(C)	[B]	[A]	—	—	—	—	[B]	************	0011101111210002	0011001110010001	
Cycle 16:	(C)	(C)	(R)	[C]	[A]	[A]	(C)	(C)	(R)	[C]	[R]	[A]	—	—	—	—	*******	0002100100110001	2211101100100000	
Cycle 17:	(B)	(B)	(A)	(A)	[R]	[R]	(C)	[R]	(C)	[R]	(S)	[R]	—	—	[C]	(S)	**********	2111001100100011	0010002111010010	
Cycle 18:	(C)	(C)	(R)	(R)	[C]	[C]	[C]	[C]	[C]	—	[S]	—	—	—	—	[S]	********	0000011110200011	0000221010000010	
Cycle 19:	(R)	(R)	(C)	(C)	[B]	[B]	[A]	[A]	—	—	(S)	—	—	—	—	(S)	*******	0000111100200001	0011001100000000	
Cycle 20:	(C)	(C)	[R]	[R]	[C]	[C]	—	—	—	—	(S)	—	—	—	—	(S)	****	0011000000100001	0100011100000000	
Cycle 21:	(S)	[R]	[C]	[C]	[R]	(S)	—	—	—	—	[S]	—	—	—	—	[S]	****	0100010000100001	0011100000000000	
Cycle 22:	[C]	[C]	—	—	—	—	—	—	—	—	[S]	—	—	—	—	[S]	***	1000000000100001	0100100000000000	
Cycle 23:	[S]	—	—	—	—	[S]	—	—	—	—	—	—	—	—	—	—	**	1000010000000000	0000000000000000	

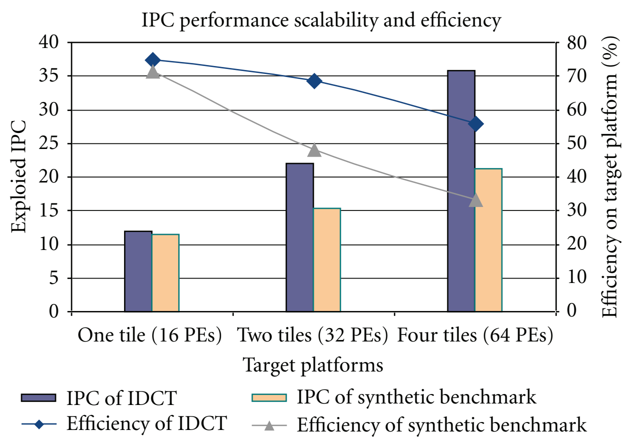


Figure 8: Performance scalability for IDCT and a synthetic benchmark.

bypassing registers that are dedicated to data relay as it does with non-transient copy disabled.

Table 3 compares the performance results for other applications with the feature of non-transient copy on and off. On average, using non-transient copy can improve the IPC by 18%. Two special cases where the IPC does not improve much are idct and sad16_c. This is because these two algorithms are designed in the manner of "in-place-update," in which an instruction replaces the source register with the new result immediately after computation. With this special coding style, registers are used very locally and few operands can be reused. In contrast, the other applications make significant reuse of operands. For example, the computation of two adjacent pixels usually can share a large amount of reference pixels. As a result, distributing the shared reference pixels to multiple PEs as non-transient copies helps for data reusing, and the performance improves significantly.

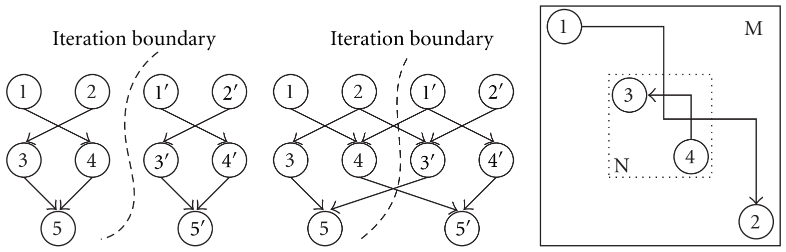
From Table 3, we can also observe that the total usage of inter-PE links is reduced by 15% on average, which can directly be translated into power saving. Moreover, for most of the applications, both the average and peak bypassing register usage are smaller when non-transient copy is enabled.

4.4. Performance Scalability Study. It can be observed from the results in Section 4.2 that the benchmark applications achieved high IPC on both small configuration and large configuration. Actually these video applications can achieve very good performance scalability because of the weak data dependency between each of their kernel iterations. This is illustrated in Figure 9(a), where intra-iteration data dependency is strong while inter-iteration data dependency is weak. As a result, these almost independent iterations can be scheduled across the whole reconfigurable array without incurring much performance penalty. However, it would be interesting to study the performance scalability if an application does not possess this weak inter-iteration property. Unfortunately, good kernels are more than often written in loop fashion with weak inter-iteration data dependency. So we have to create a somewhat contrived synthetic benchmark to simulate what would happen if a kernel has strong inter-iteration dependency. This is done by generating instructions with data dependency with each other for the synthetic benchmark. In order to compare the performance scalability difference between a typical IDCT (row+col) with weak inter-iteration dependency and the synthetic benchmark, we produce the synthetic benchmark to contain the same number of instructions of the IDCT and similar level of instruction-level parallelism but with strong inter-iteration dependency as illustrated in Figure 9(b).

The performance results are plotted in Figure 8 for compiling these two applications on three configurations: one tile, two tile, and four tiles. It is clear that when the target platform has more PEs, IDCT scales better than the synthetic benchmark though they both get performance boost with more available PEs. Specifically, in four-tile configuration where 64 PEs are available, >35 IPC can be achieved in the case of IDCT, but only 21 IPC can be achieved for the synthetic benchmark. Another interesting point about the CGRA efficiency can be observed from Figure 8. Both applications experience efficiency drop when the number of available PEs becomes larger. This is because scattering an application across a large CGRA tends to introduce more operation "bubbles" in the final execution trace. Therefore, the efficiency may degrade with the increasing of PE numbers.

For the synthetic benchmark, not only that the performance does not scale as well as IDCT but also that the efficiency drops more than IDCT. We think this is mainly due to the difference between the data dependency patterns in these two applications. Since IDCT has weak inter-iteration dependency the operations only need to talk to other operations in the same iteration on neighboring PEs locally. In contrast, the strong inter-iteration data dependency pattern in the synthetic benchmark requires more global communication from one PE to another in a different tile. Therefore, the larger the scale of the PE array, the longer distance for transferring data between remote PEs. Consider a scenario in Figure 9(c) for the synthetic benchmark. Assume M is the computation grid (analogous to a metropolis area) and N is a subgrid within M (the downtown). Two routing requirements are shown. There can be chances that the path connecting node 1 and node 2 will also make use of channels in the subgrid N (downtown). Accumulating the effects, the N (downtown) could have heavy communication (traffic) jam. To avoid congestion, the long routing path will probably be dispersed to the outskirt area. Since long distance across multiple tiles may need more routing cycles, the overall performance will be dragged down by this *downtown effect*.

This suggests that it may not be always a good practice to scatter an application to as many available PEs as possible. For certain applications, instead of having the whole computational array to be mapped for a single application kernel, it is preferable to partition the tile-based FDR-CGRA into several regions. Each region carries out one application and run multiple parallel applications simultaneously. In the synthetic benchmark case, compiling it onto one tile and save the other tiles for other tasks may yield the highest efficiency.



(a) Weak inter-iteration dependency (b) Strong inter-iteration dependency (c) Downtown effect

Figure 9: Different benchmark DFGs and downtown effect.

5. Conclusions

In this paper, we proposed FDR (fast data relay) to enhance existing coarse-grained reconfigurable computing architectures (CGRAs). FDR utilizes multicycle data transmission and can effectively reduce communication traffic congestion, thus allowing applications to achieve higher performance. To exploit the potential of FDR, a new CAD inspired compilation flow was also proposed to efficiently compile application kernels onto this architecture. It is shown that FDR-CGRA outperformed two other CGRAs, RCP and ADRES, by up to 21% and 29%, respectively.

References

[1] S. Hauck and A. DeHon, Eds., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*, Morgan Kaufmann, Boston, Mass, USA, 2007.

[2] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings—Computers and Digital Techniques*, vol. 152, no. 2, article 193.

[3] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12–21, April 1997.

[4] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the The 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 225–235, June 2000.

[5] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the 16th Asia South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 564–570, 2001.

[6] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD–reconfigur-able pipelined datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*, 1996.

[7] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[8] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[9] R. W. Hartenstein and R. Kress, "Datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '95)*, pp. 479–484, September 1995.

[10] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pp. 157–166, April 1996.

[11] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, pp. 1224–1229, February 2004.

[12] O. Colavin and D. Rizzo, "A scalable wide-issue clustered VLIW with a reconfigurable interconnect," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 148–158, November 2003.

[13] M. B. Taylor, W. Lee, J. Miller et al., "Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ILP and streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pp. 2–13, June 2004.

[14] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 191–200, February 2009.

[15] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. S. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 166–176, October 2008.

[16] G. Lee, K. Choi, and N. D. Dutt, "Mapping multi-domain applications onto coarse-grained reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 637–650, 2011.

[17] T. Suzuki, H. Yamada, T. Yamagishi et al., "High-throughput, low-power software-defined radio using reconfigurable processors," *IEEE Micro*, vol. 31, no. 6, pp. 19–28, 2011.

[18] Z. Kwok and S. J. E. Wilton, "Register file architecture optimization in a coarse-grained reconfigurable architecture," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 35–44, April 2005.

[19] S. Cadambi and S. C. Goldstein, "Efficient place and route for pipeline reconfigurable architectures," in *Proceedings of*

the International Conference on Computer Design (ICCD '00), pp. 423–429, September 2000.

[20] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA '00)*, pp. 375–386, January 2000.

[21] R. Balasubraamonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture (ACM/IEEE '01)*, pp. 237–248, December 2001.

[22] B. Mei, F. J. Veredas, and B. Masschelein, "Mapping an H.264/AVC decoder onto the adres reconfigurable architecture," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 622–625, August 2005.

[23] C. Lattner, "Introduction to the LLVM Compiler Infrastructure," in *Itanium Conference and Expo*, April 2006.

[24] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, chapter 3, Morgan Kauffmann, Boston, Mass, USA, 4th edition, 2006.

[25] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[26] R. Nair, "A simple yet effective technique for global wiring," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 2, pp. 165–172, 1987.

[27] Xvid video codec, http://www.xvid.org/.

[28] Opensource H.264 reference code, http://iphome.hhi.de/suehring/tml/.