An FPGA-based Heterogeneous Coarse-Grained Dynamically Reconfigurable Architecture

Ricardo Ferreira
Departamento de Informatica
Universidade Federal de Vicosa
Vicosa, Brazil
ricardo@ufv.br

Julio Goldner Vendramini
Departamento de Informatica
Universidade Federal de Vicosa
Vicosa, Brazil
julio.vendramini@ufv.br

Lucas Mucida
Departamento de Informatica
Universidade Federal de Vicosa
Vicosa, Brazil
lucas.mucida@ufv.br

Monica M. Pereira
Instituto de Informatica-PPGC
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
mmpereira@inf.ufrgs.br

Luigi Carro
Instituto de Informatica-PPGC
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
carro@inf.ufrgs.br

ABSTRACT

Coarse-grained reconfigurable architecture has emerged as a promising model for embedded systems as a solution to reduce the complexity of FPGA synthesis and mapping steps, consequently reducing reconfiguration time. Despite these advantages, CGRA usage has been limited due to the lack of commercial CGRA circuits. This work proposes a virtual and dynamic CGRA implemented on top of an FPGA. This approach allows the usage of commercial-off-the-shelf FPGA devices combined with the advantages of CGRAs. The proposed architecture consists of a set of heterogeneous functional units (FU) and a global interconnection network. The global network allows any FU to be used at each cycle, which reduces significantly the placement complexity. In addition, we introduce a polynomial mapping algorithm which includes scheduling, placement and routing steps (SPR). Moreover, the proposed approach performs a very fast placement and routing in comparison to similar CGRA approaches. The three SPR steps are computed in few milliseconds. The feasibility of this approach is demonstrated for a suite of digital signal processing benchmarks.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Adaptable architectures, Dataflow architectures, Heterogeneous systems

General Terms

Performance

CASES'11, October 9–14, 2011, Taipei, Taiwan.

Keywords

Reconfigurable Architectures, CGRA, FPGA, Placement, Routing, Scheduling, Interconnections, Multistage

1. INTRODUCTION

As scaling continuously increases circuit densities, increasing the amount of resources is no longer a challenge in terms of available area and cost [5]. As a consequence, many solutions are proposed to take advantage of abundant resources and increase device's efficiency. Spatial computing emerges as a solution for increasing performance by distributing computations in space rather than in time [8]. In fact, increasing the number of parallel processing elements allows concurrent operation and consequently accelerates computation.

Reconfigurable computing emerges as an alternative to reduce the time-to-market, and at the same time adds flexibility and fast prototyping for spatial and/or temporal computing [16, 10, 7]. Current FPGA devices provide flexibility by having a large number of fine-grained reconfigurable units and interconnection elements. However, one of the main challenges consists in mapping generic applications onto these complex FPGA devices. The problem is NP-complete and the current synthesis and mapping tools are CPU time-consuming [16, 25]. In fact, performing placement and routing requires long time that can be in order of minutes, hours or, in the worst cases, days. This bottleneck has been one of the main challenges that prohibit the widespread use of FPGAs.

Coarse-grained reconfigurable architectures (CGRA) are reconfigurable at word level (16 bits, 32 bits, etc.), while FPGAs are reconfigurable at bit level. The direct consequence of working at word level is the reduction on the number of configuration bits; the amount of time to configure; and the placement and routing complexity [12]. However, even for CGRA, the placement and routing is a NP-complete problem for spatial computation. In addition, when temporal computing is considered, the scheduling problem is also a NP-complete problem.

Despite the advantages of CGRA, there is a lack of compiler tools and a lack of commercial devices. Most tools are specific to a subset of applications and specific architecture. Therefore, since few CGRA commercial devices are available [7], an alternative is to implement CGRA as virtual devices on top of commercial-off-

the-shelf FPGAs. The main advantage of this approach is reducing the complexity of handling fine-grained FPGAs architectures, such as in placement and routing steps, consequently providing a more efficient way to configure the architecture by reducing the time overhead introduced on these steps.

In this context, this work proposes an FPGA-based CGRA. This approach offers portability, since the virtual coarse-grained architecture can be implemented on top of any commercial off-the-shelf FPGA. Moreover, the proposed approach also allows fast prototyping, as it implements a simpler configuration algorithm when compared to fine-grained architectures. The proposed architecture consists of a set of word level functional units, such as adders and multipliers, and a global interconnection network. The global network simplifies the complexity of scheduling, placement and routing steps to implement spatial and temporal computing. The global network consists of two parallel blocking multistage interconnection networks (MIN).

A n input/output MIN has the cost complexity $O(n\log(n))$ in comparison to crossbar networks which have a prohibitive cost of $O(n^2)$, even for small values of n. Moreover, while most MIN approaches propose to use rearrangeable MINs with $2\log(n)-1$ stages, we propose to use extra level blocking MINs. The proposed MINs have $\lg(n)+k$ stages, where k is the number of extra stages or levels, and $k \leq \log(n)-1$. In addition, several applications have multicast connections. In these cases, a rearrangeable MIN should have $4\log(n)-2$ stages. Our MIN has at most $2\log(n)-1$ stages, which reduces the latency. Experimental results show that the routing approach is efficiently even using blocking network.

To evaluate the proposed architecture we developed a novel scheduling and mapping tool. Our approach is based on a simple greedy algorithm, and the reported CPU time of around 10-100 milliseconds is 10 to 100 times faster in comparison to another CGRA works [16, 7, 25]. Therefore, our approach could be included in just-in-time compilers or even reconfiguration at execution time.

Our architecture simplifies the scheduling and placement by using a global network where any functional unit (FU) could reach any FU in one cycle. The MIN routing algorithm is faster than traditional algorithms with polynomial complexity $O(n\log(n))$. More details on how the routing algorithm can be implemented can be found in [9]. We also perform the three steps together: scheduling, placement and routing (SPR). The experimental results show a high number of instructions per cycle (IPC) as well as a scheduling density, which is the average number of active FUs per cycle.

The remainder of this paper is organized as follows. Section 2 presents the proposed architecture and the global interconnection network. Section 3 details the scheduling, placement and routing algorithm. Section 4 presents experimental results. Some related works are presented in Section 5. Finally, conclusions and future works are presented in Section 6.

2. ARCHITECTURE

Fig. 1 depicts the architecture structure which consists of three parts. First, there is a set of registers. The register outputs are directed connected to a set of functional units (FUs). At each clock cycle, the data is sent from the register and processed in the FUs. A FU could be an adder, a multiplier, or even a single wire in case of bypassing unit. The FU outputs are connected to a global interconnection network. The results generated by the FUs are sent back to the registers to be used in the next clock cycle. The use of a global network significantly simplifies the communication model, and consequently the mapping tool. Compared to other topologies used in CGRAs, such as two dimensional meshes [16, 21, 22] or stripes [11], the global structure has the advantage of being able to support structured as well as unstructured communication patterns. This is important since different applications have different communication patterns. Even a single application could have several different communication patterns.

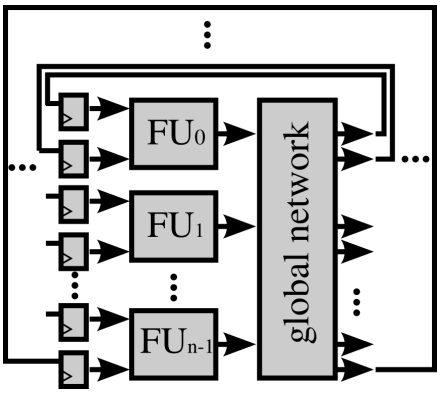


Figure 1: General Architecture.

2.1 Example

This section presents an example of using the proposed architecture to map a simple dataflow graph. For better visualization, the Fig. 1 is redrawn by placing the registers in last position as shown in Fig. 2(b-d). Fig. 2(a) depicts the dataflow graph used in this example, where the nodes represent the operations and the edges implement the data-dependency relationship between the operations. Let us suppose a simple architecture which has only two functional units. One mapping possibility is depicted where three configurations are used. Each configuration is executed in one cycle. Starting the execution, nodes A and B are placed in the first configuration as shown in Fig. 2(b). After the execution, the results are routed through the global interconnection network to the next configuration. Therefore, as shown in Fig. 2(c), the values of A and B are stored in the register file. In the second cycle the operation of nodes C and D are placed, the results are computed and sent to the register file through the global interconnection network to be used in the next cycle. Finally, the operation E is computed in the third cycle by using the values of C and D read from the registers.

The architecture is flexible and several scheduling and mapping strategies could be implemented. Section 3 will present a pipeline scheduling and mapping.

2.2 Functional Units

The FU could be homogeneous or heterogeneous as the proposed architecture uses a global communication approach. Most of proposed CGRAs use homogeneous FUs [16, 21, 22, 11]. This assumption simplifies the scheduling, the placement and the routing steps, since any operation can be mapped onto any FU. However if multipliers and/or memory ports are included in all FUs, the CGRA costs could be prohibitive. Heterogeneous FU could reduce cost, power and implementation complexity [25]. Nevertheless, most CGRAs assume 2-D mesh topologies [16, 21, 22] which increases the complexity of placement and routing of heterogeneous resources.

2.3 Dynamic Interconnection Network

The crossbar network has almost the ideal network properties: non-blocking routing, multicast, and straight-forward routing algorithm. However, the cost complexity is $O(n^2)$ which is prohibitive even for small values of n. We propose to use multistage intercon-

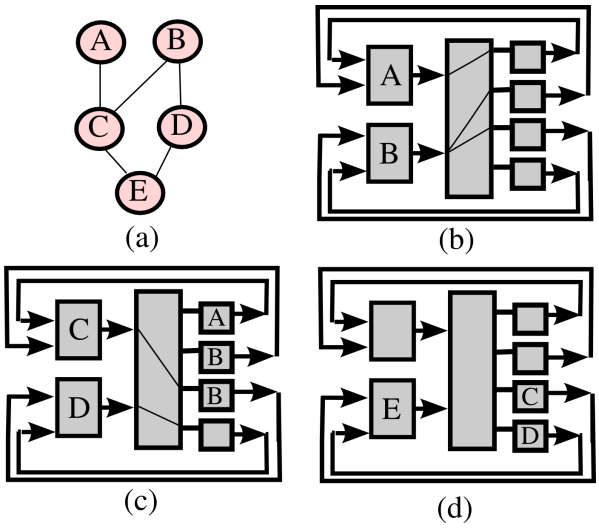


Figure 2: A Simple Mapping Example: (a) DataFlow Graph; (b) First Configuration; (c) Second Configuration; (d) Third Configuration.

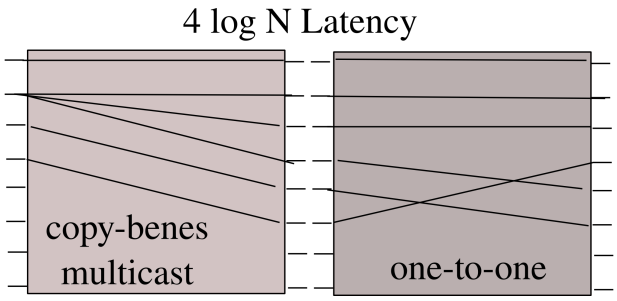


Figure 3: Copy Benes for Multicast

nection networks (MINs) which the cost complexity is $O(n\log n)$ and it can be efficiently implemented on FPGAs [18, 20, 24, 9].

MINs have been studied since 1953, when Clos network was introduced [6]. A MIN can be non-blocking, rearrangeable or blocking. A non-blocking MIN can establish all connections in any order. A rearrangeable MIN can also establish all connections. However, the routing algorithm should know a priori all connections, and a given order should be followed.

A Benes network [3] is the most studied rearrangeable network. It has $2\log(n)-1$ stages. However, in case of multicast connections, where one input is connected to more than one output, Benes cannot be rearrangeable. To solve this, one approach is to use two Benes network in series [17]. This approach is named copy-benes. The first network maps all multicast connection in successive set as shown in Fig 3. The second network performs a one-to-one permutation. As a Benes is rearrangeable for a one-to-one permutation, all connections are realized. However, there are $4\log(n)-1$ stages, and the latency is the double of a single Benes.

We propose the use of two blocking networks in parallel. Since a blocking network has only $\log(n)$ stages, the proposed approach could reduce significantly the network latency. A MIN is blocking if it is not possible to establish all connections. Omega, butterfly and baseline are the most studied blocking MINs. In this work, the Omega network will be used. However other class equivalent MIN could be used.

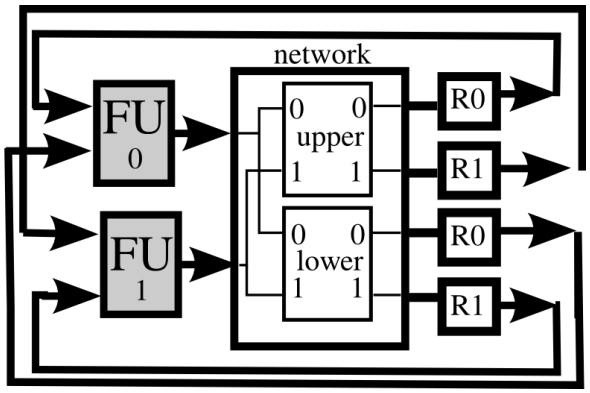


Figure 4: Parallel Omega Network

The proposed MIN is based on previous work [9]. The number of MIN stages decreases as the switch radix increases, and it is proportional to $\log_r$ where r is the radix. However, the switch complexity increases quadratically. The best radix size depends on the technology and mapping tools. We have shown that Radix 4 MINs can be efficiently implemented on top of 6 input LUTs FPGAs. As a result, the r_4 MIN is approximately half the cost of a r_2 network of identical capacity. The previous analysis [9] has also shown that the two parallel blocking Omega MIN are able to route as a non-blocking network, even in presence of multicast permutations.

There is only one path for each input/output pair in an Omega MIN. By adding extra levels, the number of path could be increased. Each extra level doubles the number of possibilities. The proposed network is based on two parallel Omega networks plus extra levels. As we will show later, there are few routing conflicts even using a blocking network.

Fig. 4 depicts a simple example of two FUs and the proposed interconnection network. The FU_0 is connected to the upper and the lower Omega network at input 0, as well the FU_1, which is connected to the networks at input 1. If there is n FUs, the FU_i is connected to the input i of both Omega networks. The output 0 of the upper Omega is connected to the first input of the FU_0. The output 0 of the lower Omega is connected to the second input of the FU_0. The upper Omega will route the first operand to the FUs and the lower network will route the second operand. If we want to connect the FU_i to the FU_j, a connection requesting $i \rightarrow j$ is generated. The connection could be routed either by the upper or the lower network. As most operators are symmetrical, if there is a routing conflict in upper Omega, the lower Omega could be used. In case of a subtractor or a divider, only one network could be used for a given operand.

Let us consider the following computation: $x = a+b, y = c*d, w = x+x$, and $z = x*y$. Fig. 5 shows a possible mapping in two steps. Suppose FU_0 is an adder and FU_1 is a multiplier. First, x and y are computed. The operations x and y are placed in FU_0 and FU_1, respectively. The values of x and y should be routed to w and z. As w is an addition, it will be placed at FU_0, and z is a multiplication and it will be placed at FU_1. As the value of x should be routed twice to w and once to z, the connections $0 \rightarrow 0$ and $0 \rightarrow 1$ from the upper network and $0 \rightarrow 0$ from the lower network will be requested, as well as the connection $1 \rightarrow 1$ from the lower network to route the value of y to z. A possible routing configuration is illustrated in Fig. 5.

Fig. 6 depicts a detailed view of MIN interconnection routing. A 4- node operation dataflow is shown in Fig. 6(a). Let us consider

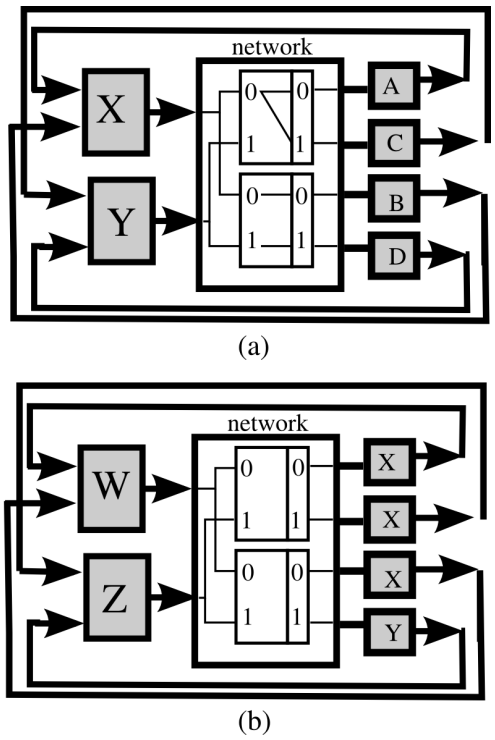


Figure 5: FU Multicast Routing: (a) First Configuration; (b) Second Configuration.

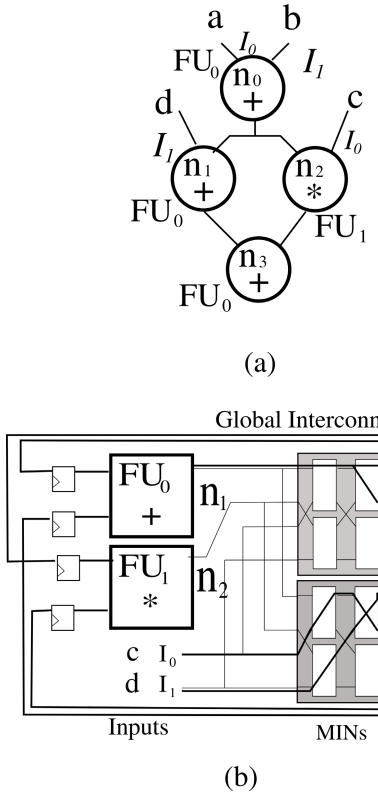


Figure 6: Detailed view of the Global Interconnection Model: (a) Dataflow Graph; (b) The Routing for the Second Configuration.

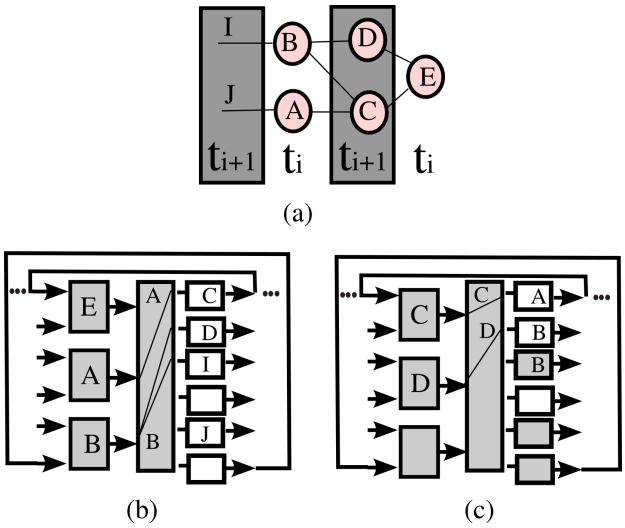


Figure 7: A Pipeline Example: (a) Dataflow and Scheduling; (b) First Configuration; (c) Second Configuration.

a simple architecture which consists of only two functional units as shown in Fig. 6(b). At least three configurations are needed to implement the dataflow graph due to the data dependence without pipeline. Let us suppose that the dataflow node n_0 has been mapped in FU_0 at the first configuration. The routing paths for the second configuration are displayed in Fig. 6(b). The value of n_0 is multicast to n_1 and n_2. For ease of explanation, a Radix 2 MIN is depicted. There are two parallel Omega networks where each one has 4 input/output. The dataflow external inputs c and d are also routed to the nodes n_1 and n_2.

The architecture is described by a set of parameterized VHDL files. The user could specify the number and type of FUs, the number of MIN extra level as well as the data width. These parameters are used by the mapping tool which will be introduced in next section.

3. MAPPING

3.1 Scheduling

A modulo scheduling approach [19] is used where the same schedule is repeated at regular intervals for each loop iteration. The constant interval between two successive iterations is referred as initiation interval (II). The minimal initial interval is a lower bound of II. The lower bound is determined from the resource requirements. The minimal number of functional units are evaluated at placement step and the interconnection resources at routing step.

Although modulo scheduling is a well-known family of algorithms since the eighties years, an optimal solution is computationally expensive. We propose an efficiently (polynomial complexity) heuristic approach. First a ASAP/ALAP scheduling is performed to determine the scheduling range of each operation. The insertion of a register is performed in unbalanced paths. Then the placement and routing are performed in a single step. The II is initialized to the minimum value based only on the minimal number of functional units. If the mapping fails, larger values of II are successively assigned until all operations have been mapped. As we will show in section 4, our approach is quite fast. In addition, more elaborate scheduling techniques could be included.

For easy of explanation, let us consider the previous example il-

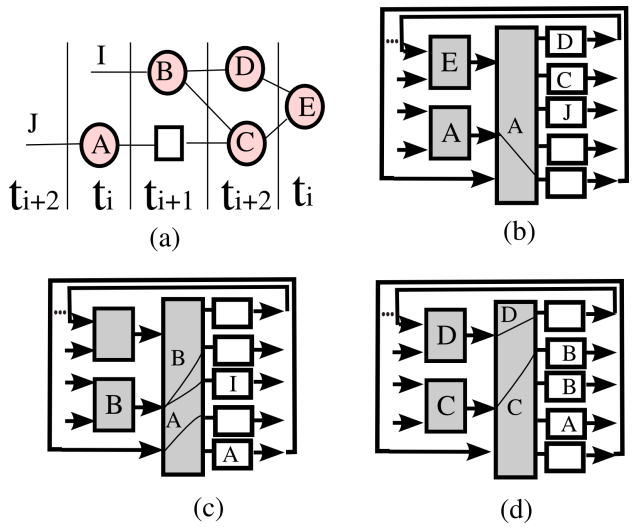


Figure 8: A Bypassing Register Example: (a) Dataflow and Scheduling; (b) First Configuration; (c) Second Configuration; (d) Third Configuration.

```
 Inputs: Dataflow Graph G, Architecture A
1 Asap_Alap(G);
2 II = minimum_resources(G,A);
3 mapping = false;
4 Cfg = II.get_first();
5 While ( ! mapping ) {
6   For each level L in G from Outputs do
7   {
8     while ( L.not_empty() )
9     {
10       n = L.remove_node();
11       if ( ! place_route(n,Cfg) )
12          fail;
13    }
14    if (fail) break;
15    Cfg = II.get_next();
16  }
17  if (fail) Cfg = II.increase();
18  else mapping = true;
19} // mapping
```

Figure 9: Scheduling Placement and Routing - SPR

lustrated in Fig. 2 with some additional assumptions. First, suppose that the architecture has three FUs as shown in Fig. 7. Second, external signals are included to insert input data in nodes A and B. As the number of nodes is greater than the number of FUs, at least two configurations are needed and the minimum II is 2. Fig. 7(a) depicts the dataflow graph. At cycle t_i, nodes A and B are computed and placed in the second and third FU (see Fig. 7(b)). The results are routed through the interconnection network to be used in the next cycle. At cycle t_{i+1}, nodes C and D receive data from A and B as input registers, and the computed results are routed through the interconnection to the first FU, where the node E will be placed at the cycle t_i. Therefore nodes A, B and E are placed in configuration t_i and nodes C and D in configuration t_{i+1}. We will explain later the signal I, J and the output data generated by node E. Although nodes share the same configurations, data have different time instants. While E is at time t_0, C and D are at time t_1, A and B are at time t_2, and finally I and J are at time t_3, for instance.

Although the latency is four cycles, the throughput is two cycles. In case of smaller architectures, more configurations will be needed. Suppose the previous example mapped onto a two FU architecture as illustrated in Fig. 8. In addition, this architecture has one register. As it is not possible to place A, B and E in the same configuration, the scheduling inserts a register to shift a node. Assuming node A is shifted. Nodes A and E are computed in configuration t_i, nodes C and D are computed in configuration t_{i+2}, and the bypassing register and node B are computed in configuration t_{i+1} (see Fig. 8). The latency increases to five cycles and the throughput increases to three cycles.

The pseudo code for the proposed SPR algorithm is depicted in Fig. 9. First the ASAP and ALAP scheduling are computed. Then, minimum II is calculated based on FU resources. A first configuration is initialized. The mapping is computed in lines 5 to 19. From the outputs through the inputs, the dataflow graph is traversal by using ALAP level. Each level is mapped in the current configuration. At line 10, a node n is removed from current level, and the placement and routing are performed. If it is not possible to perform neither the placement nor the routing, the value of II is incremented and a new mapping is computed. As the number of configuration is increased, the number of resources per configuration will be reduced and the mapping for the new value of II is possible.

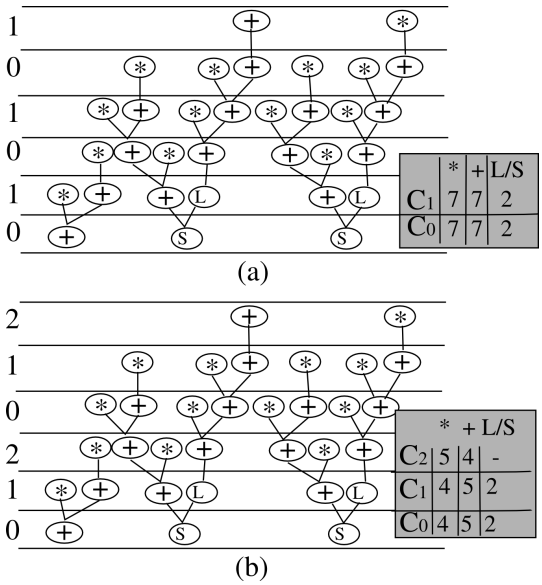
Fig. 10 shows a dataflow graph extracted motion vector and two scheduling configurations. Suppose that the initial value of II is 2. The first configuration c_0, ordered by ALAP level, has seven adders, seven multipliers and two load operators. The second configuration has similar values, seven adders, seven multipliers and two store operators, as shown in Fig. 10(a). The maximum number of adder or multiplier per configuration is 7. If II is increased to three, the number of resources per configuration decreases. The first configuration c_0 has 5 adders, 4 multipliers and 2 store operators. c_1 has 5 adders, 4 multipliers and 2 load operators, and c_2 has 4 adders and 5 multipliers. The maximum number of adder or multiplier per configuration is 5, as depicted in Fig. 10(b).

3.2 Placement

This previous example shows a case where the number of operators is well-distributed and balanced between the configurations. Now, considering the dataflow for a finite impulse filter (FIR) displayed in Fig. 11. Suppose that the target architecture has only 8 adders, 6 multipliers and 12 input/output units; the dataflow graph has 10 adders, 11 multipliers and 23 input/output units. Therefore at least 2 configurations are needed. For two configurations, there are 2*8=16 adders, 2*6=12 multipliers, and 2*12=24 input/output units. By using the ALAP order without rescheduling, c_0 will need 19 input/output units and c_1 will need 7 multipliers as shown in Fig. 11(a).

The SPR algorithm starts from the outputs toward the inputs. The scheduling, placement and routing of the graph onto target architecture are executed for each node until there are no more available FUs in current configuration or a routing conflict is found.

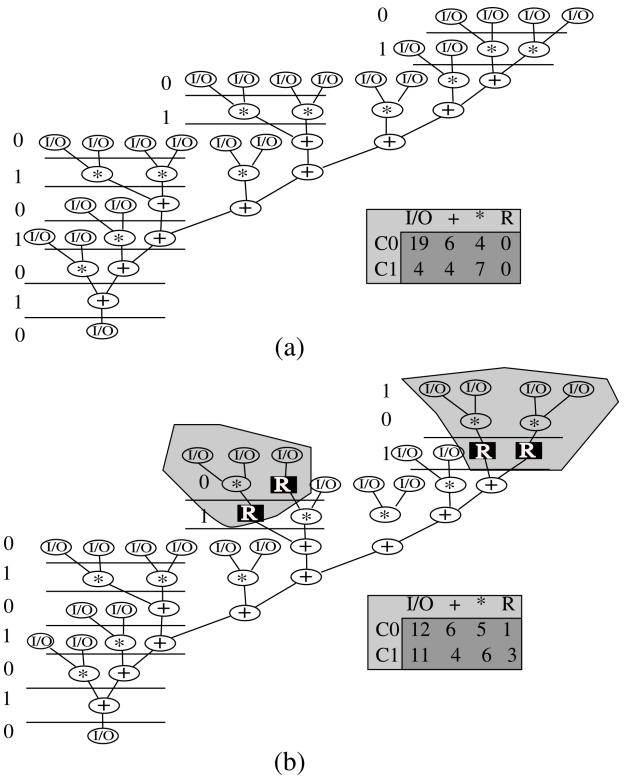
The placement algorithm is described in lines 1-6 (see Fig 12). At line 1, the function *Cfg.get_FU* returns a free FU for a given operation. If there are no more FUs, the placement will insert a register unit. This case will happen when the seventh multiplier is requested for the configuration 1 as shown in Fig. 11(b). A register will be inserted and routed at configuration 1, and the multiplication will be rescheduled to configuration 0, as well as their input descendents. The SPR will continue until two more multipliers are



	*	+	L/S
C_1	7	7	2
C_0	7	7	2

	*	+	L/S
C_2	5	4	-
C_1	4	5	2
C_0	4	5	2

Figure 10: Two Scheduling of Motion Vector: (a) Two Step Scheduling; (b) Three Step Scheduling.



	I/O	+	*	R
C0	19	6	4	0
C1	4	4	7	0

	I/O	+	*	R
C0	12	6	5	1
C1	11	4	6	3

Figure 11: FIR dataflow and Resource Allocation: (a) ALAP order; (b) SPR.

```
 place_route (node n, Config Cfg) {
1   FU = Cfg.get_FU(n);
2   if ( FU = null ) {  // Missing FU
3     FU = Cfg.get_register();
4     if ( FU = null ) return false;
5     reschedule(n);
6   }
7   return route(n,FU,Cfg);
8 } // end place
9
10 route(node n, unit FU, Config Cfg) {
11    out = n.get_output();
12    FU_out = out.get_FU();
13    successful  = net_routing(FU,FU_out);
14    if (! successful ) {
15      FU = Cfg.get_register();
16      if ( FU = null ) return false;
17      reschedule(n);
18      n.set_FU(FU);
19      successful  = net_routing(FU,FU_out);
20    }
21    return successful;
22 } // end route
```

Figure 12: Placement and Routing Algorithm

requested for configuration 1, then two registers will be inserted and the operations will be moved to configuration 0. As an indirect consequence, when the multiplications are moved to configuration 0, I/O operators are moved to configuration 1. Four registers are included, and the latency will increase one cycle. Nevertheless, the throughput is two cycle and a high degree of parallelism is achieved, since most FUs are used in all cycles.

If there are no more registers, the mapping will fail. As mentioned before, if the mapping fails, then II is incremented, and a new mapping with II+1 configurations will be computed.

3.3 Routing

The routing is integrated in scheduling and placement algorithm in the proposed SPR approach. The pseudo code is described in lines 10-21 (see Fig 12). The scheduling and placement scan the dataflow from the output to the input. At a given level, when a FU is placed, we need to get the FU output to perform the routing (lines 11-13). The function *net_routing* tries to route the FU for the current configuration to the output FU in the previous configuration. For ease of explanation, the code of this function is not detailed. If there is a conflict for the current FU, others compatible FUs are tried. If there is any compatible FU which could be connected to the output FU, the routing will fail. In this case, the algorithm tries to insert a register (lines 14-20). If it is not possible to route any register or there are no registers available, the configuration mapping will fail, and II will be incremented.

As mentioned in Section 2.3, the interconnection network is based on multistage networks. The proposed network consists of two parallel Omega networks. Suppose that the current node is placed on the FU_i in configuration a, and the node output is placed on the FU_j in configuration $a-1$. Two parallel routing requests will be generated to verify if it is possible to route $i \rightarrow j$ in upper and lower Omegas (see Fig. 4).The upper Omega will be used first, in case of both network are routable. The Omega routing algorithm is quite simple, the complexity for each routing connection is $O(\log(n))$. Therefore, the proposed approach integrates the SPR steps in a sin-

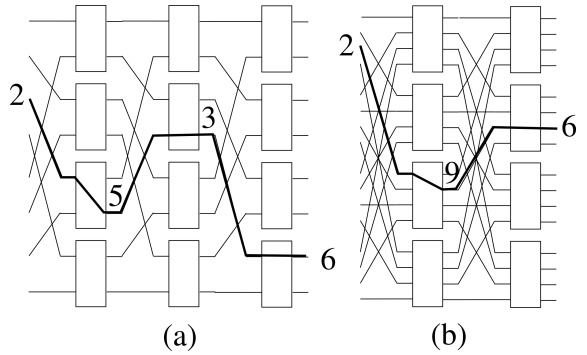


Figure 13: Omega Routing Example: (a) Radix 2; (b) Radix 4.

gle step. This is a heuristic approach. The main advantages are the fast CPU time to integrate this algorithm in just-in-time compilers or even in dynamic reconfigurable systems.

The net_routing(I,O) algorithm can be implemented in software or hardware [9]. The algorithm will find a path from the MIN input I to the MIN output O. Let us consider I and O the $\log n$ binary representation for the input/output addresses. Fig. 13 shows a Radix 2 and a Radix 4 Omega network. There is a unique path for each I/O pair in an Omega MIN. Let us suppose $n = 8$ and we would like to connect the input 2 to the output 6 for the Radix 2 MIN depicted in Fig. 13(a). The path will be determined by a routing word W, which is the concatenation of I and O, for our example $W = 010110$. The line after the first stage is determined by a $\log n$ bit window inside the routing word, starting at second bit for a Radix 2 MIN. The window is shifted right one bit after each stage. For our example, the windows are 0 101 10, 01 011 0 and 010 110 . Therefore the path passes through the lines 5,3 and 6 as shown in Fig.13(a). Let us suppose $n = 16$, for a Radix 4 MIN, the window starts in the third bit and shifts two bits per stage. For our example, the windows are: 00 1001 10 and 0010 0110 (see Fig.13(b)).

4. EXPERIMENTAL RESULTS

We have tested the proposed approach on a set of dataflow graphs available in [2]. These graphs were carefully selected from over 1400 data flow graphs of the Mediabench benchmark. The dataflow graph characteristics are shown in Tab. 1. Columns I/O, *Reg*, M, $+$, $*$, L show the number of input/output, registers, memory operations, adders, multipliers and logical operators, respectively. Since several dataflow graphs available in [2] have adders or multipliers without any input signals, we have included two input nodes for each external operator. Column *Reg* shows the number of added register to balance the multicast edges for pipeline execution.

The *collapse* benchmark is the largest basic block of a quadrature mirror filterbank. The *feedback* benchmark takes in a vertex buffer and calculates texture coordinates for a feedback buffer. The *fir* and the *fir1* are two finite input response filters. The *h2v2* is a core part of the jpeg compression algorithm. The *Inter* performs linear interpolation between two points. The *matmul* multiplies two 4x4 matrices. The *invert* is a matrix inversion routine. The *jpeg_slow* is a forward discrete cosine transform benchmark. The *jpeg_fast* is an inverse discrete cosine transform benchmark. The *smooth* consists of a basic block of four unrelated computations, which can all be run in parallel without any worries about data dependencies. Finally, the *writebmp* has a high level of parallelism with 106 nodes and 88 edges, the depth is never more than 7.

Tab. 2 shows six architecture configurations. These architecture

Table 1: Dataflow Characteristics

name	I/O	Reg	M	+	*	L
collapse	12	40	18	29	9	12
feedback	42		11	23	18	6
fir	23			10	11	
fir1	17			15	8	
h2v2	36	10	17	32	3	4
Inter	96		16	56	36	4
matmul	50		24	45	40	5
invert	154	24	80	106	141	22
jpeg_slow	52	100	24	78	37	16
jpeg_fast	54	253	24	78	37	16
smooth	130	8	48	80	69	9
writebmp	76	6	35	37	2	57

Table 2: Target Architecture Characteristics

name	Size	+	*	L	M	I/O	R	Total
A1	64	10	10	5	5	16	18	30
A2	64	18	8	4	4	12	18	34
A3	64	10	8	4	4	20	18	26
A4	256	48	48	28	28	64	40	152
A5	256	60	32	26	26	72	40	144
A6	256	48	32	20	20	96	40	120

configurations were chosen to evaluate the SPR results considering different number of functional units; I/O ports; adders; multipliers; logical operators; load/stores; and registers. Since each of these elements can influence the SPR results. The first column is the architecture name. Column size displays the global network I/O size. We evaluated three medium size architectures with 64 I/O MINs and three large architectures with 256 I/O MINs. We use power of 4 MIN size, since radix4 MIN has smaller area and delay than radix2 in current FPGA technology. Columns $+$, $*$, L, M, I/O, and R show the number of adders, multipliers, logical operators, load/stores, input/outputs, and registers, respectively.

Last column displays the total number of FUs without taking into account the number of registers and I/O. The architecture A_2 has more adders than A_1, and the architecture A_3 has more I/O. For the architecture A_5, the number of adders are incremented as well as the number of I/O ports in comparison to the architecture A_4. The architecture A_6 has more I/O than A_4.

Tab. 3 shows the SPR results for a subset of the dataflow graphs onto the architecture A_1 detailed in Tab. 2. The two first columns display the benchmark name and the number of nodes without considering the registers. The minimum latency and the minimum II are shown in columns *Lat* and *MinII*. The minimum II is calculated based on the FU resources. The number of instructions per cycle is displayed in column IPC, which shows how many operations are

Table 3: SPR results for Architecture A_1

name	n	Lat	MinII	IPC	II	CPU
collapse	80	10	3	19.25	4	17.63
feedback	100	9	3	25	4	23.71
fir	45	12	2	22	2	14.04
fir1	40	11	2	20	2	10.29
h2v2	90	20	4	22.5	4	49.92
Inter	208	10	6	26	8	117.32
matmul	164	11	5	23.43	7	63.18

Table 4: SPR results for Architecture A_6

name	n	lat	MinII	IPC	II	cpu time
inter	208	10	2	104	2	51.71
invert	503	13	5	84	11	59.42
jpeg_slow	196	17	3	65.33	3	**180.05**
jpeg_fast	187	20	4	37	5	**336.25**
matmul	164	11	1	82	2	59.25
smooth	336	13	3	67.2	5	180.22
writebmp	207	9	2	103	2	69.49

Table 5: Architecture Mapping on top of an FPGA Xilinx Virtex6

arch	Slices Registers	Slices LUTs	DSP	Clock MHz
A1	4864	22609	40	100
A2	4701	22545	32	100
A3	4991	22731	32	100
A4	18625	123958	192	80
A5	18881	124230	128	80
A6	19700	123457	128	80

computed in one cycle on average. If we divide the IPC by the number of FUs without taking into account the register resources, we can measure the scheduling density which is on average close to 50%. This shows the average number of active FUs per cycle. The IPC is high, ranging from 19 to 26. The proposed SPR is a greedy heuristic and the average value of II is 20% above of the minimum value of II. One of the main drawbacks of related works, presented in section 5, is the high CPU time once the problem is NP-complete. The proposed SPR algorithm requires a very small CPU time as shown in last column on 2.26 GHz Intel Core 2 Duo P8400 processor (Centrino 2). The average CPU time is 40 milliseconds.

Tab. 4 shows the SPR results for a subset of the dataflow graphs onto the architecture A_6 detailed in Tab. 2. This table is similar to Tab. 3. However, a subset of large dataflow graphs is evaluated. Large dataflow graphs could not be mapped in the previous architecture A_1 due to the lack of registers. The architecture A_6 is significantly greater than A_1. The *invert* is the largest benchmark. The value of II is the double of the minimum value. For this case, the reason is the small number of multipliers. Moreover, the A_6 has only 40 registers, and adder units could also be allocated as a register. If the architecture A_4 is used, the value of II is 6, close to the minimum value. On average, the value of II increases 40%. The IPC is very high, ranging from 37 to 104. The results show up to 100 IPC and the average value of II is 4. The average CPU time for large dataflow graphs is 130 milliseconds.

Tab. 5 displays the total of FPGA resources for each architecture configuration from Tab. 2. This results are generated for a Xilinx Virtex6 by using ISE 12.4 tool. The small architectures with 64 I/O global network uses on average 1% of FPGA register resources, 15% of LUT resources and 4% of DSP resources. The DSPs are used to implement the multipliers. The largest architecture uses on average 6% of FPGA register resources, 82% of LUT resources and 16 to 25% of DSP resources. The FPGA area is dominated by the global interconnection network. However, it is important to highlight that if two crossbars were used instead of the multistage network, the small architecture like A_1 (see Tab. 3) would require 80% of the FPGA LUT resources, and the largest architecture would not even be feasible due to the lack of LUT resources. Although the LUT area is dominated by the interconnections, as shown in the results, the proposed approach based on MINs significantly reduces the FPGA area, allowing mapping larger architectures.

Although in the heterogeneous architecture the multipliers use the dedicated DSP units and the area is dominated by the interconnections, homogeneous architectures could increase up to 30% the area costs due to the use of homogeneous FUs. However, since most CGRA approaches are based on homogeneous FUs, and few works solve the mapping problem for heterogeneous CGRA [1], to evaluate the homogeneous solution, we have also implemented our architecture with 34 homogeneous FUs for a 64 input/output network. As expected for this solution, the area increases 25% in total LUT area while the number of dedicated DSP units increases four times. Moreover, for larger architectures like A_4,A_5, and A_6, a homogeneous approach would consume all DSP and LUT resources.

As the current FPGA technology has a large number of dedicated DSP, the heterogeneous approach allows us to add float point units to the current architecture. For example, we have implemented the architecture A_1 with 8 integer multipliers and two 32-bit float point multipliers. The new architecture uses the same number of DSP and the LUT area increases only 0.5%. Even if two 64-bits float point units are added, the number of dedicated DSP units increases 25% while the area increase for a homogeneous architecture is around 4 times without any float point unit. The increase in LUT area is only 2% for two 64-bits float point units.

5. RELATED WORK

Most CGRAs are based on 2-D mesh topology [16, 10]. The ADRES SPR algorithm uses module scheduling, simulated annealing and pathfinder for scheduling, placement and routing, respectively. However, the algorithm is time-consuming as reported by the authors [16], where a 80-nodes dataflow graph could be mapped in close to 1000 seconds. The architecture has 64 homogeneous FUs in a 2-D grid topology.

A mapping tool for an adaptive CGRA is presented in [10]. The proposed tool could support a variety of CGRAs. The approach is evaluated by using a homogeneous architecture with 288 functional units distributed in 16 clusters. Each cluster has 4 ALUs, 4 I/O streams, 4 structures for hold configured constants, 2 local block RAMs. The experimental results show eight dataflow benchmarks which are scheduled and mapped. The dataflow size ranges from 157 up to 518 nodes. The II values range from 3 to 9, which are similar to our approach. However, neither the CPU time nor physical area/delay is reported.

An architecture independent mapping approach for CGRA has been proposed in [25]. As most CGRA performance is strongly dependent on specific and customizing compiler tools, [25] proposes a general approach based on graph equivalence algorithms. However, temporal mapping is not considered. The architecture should be large enough to implement the entire dataflow. Even without temporal mapping, the problem is NP-complete [25]. The experimental results use small dataflow ranges from 9 to 27 nodes. The report CPU ranges from 300ms to 800ms.

The architecture and the scheduling presented in [25] have been proposed in [13], where temporal and spatial mapping are implemented. The temporal mapping uses loop pipelining and the implementation is based on a column approach. The architecture consists of a 2-D mesh of FUs, and the operations flow column by column from left to right. The mapping CPU time is not reported in [13]. Recently, a new routing CGRA is proposed in [14], which is also based on the temporal approach presented in [13]. The routing uses Steiner pointers, and the reported CPU time is around 200ms for graphs up to 20 nodes.

A CGRA based on multistage interconnection has been presented in [23]. The architecture has six stripes. Each stripe has a cluster of 16 or 8 FUs. The output of one stripe is connected to the next stripe. Each cluster has a Benes network to interconnect the FU and I/O signals to/from the previous/next stripe. However, mapping tools are not presented. Moreover, only four benchmarks are evaluated and mapped by hand. This approach is based on bit-serial computation.

Recently, the authors of [7] proposed an FPGA-based CGRA as a virtual architecture. The goal is to ensure circuit portability for different FPGA devices and to reduce the complexity and CPU time of FPGA placement and routing steps. The CGRA has island-type topologies like a traditional FPGA. The placement is based on simulated annealing as VPR [4]. The routing is based on pathfinder [15]. The reported CPU time is on average 1.3 seconds to map DSP benchmarks ranging from 30 to 50 operators. Furthermore, the scheduling is done by hand.

Also recently, two multiprocessor architectures have been proposed based on multistage network implemented on FPGA devices [18, 20]. A global cellular automata based on multiprocessor approach is presented in [20]. The multiprocessor consists of a set of softcore NIOSII processors. Systems with up to 32 cores were implemented on an FPGA. The core has a local memory with two ports. One to connect to the local core and the other to connect to the global interconnection network. A hand-mapping application is used to validate the approach.

The multiprocessor architecture proposed in [18] consists of N miniMIPs processors, N data memories and N instruction memories. The instruction memory is local to each processor. Two multistage networks are used. One to send data from any processor to any data memory module, and another to send back data from the memories to the processors. The multistage is used on packet-switching mode. Systems with up to 16 cores were implemented on an FPGA. No mapping or compiler tools have been presented, which makes hard to test benchmarks and evaluate the time and performance results. Our approach uses MIN to interconnect a large set of FUs, while the approach proposed by [18, 20] focus on multiprocessor communication.

6. CONCLUSIONS

In this paper, we have proposed a dynamic reconfigurable coarse-grained architecture (CGRA) and a scheduling, placement and routing (SPR) approach by using pipelining techniques. The experimental results shows up to 50% of resources utilization on average per cycle. Despite the large number of reconfigurable coarse-grained that have been proposed in last two decades, few of them have physical implementations. This work presents a virtual CGRA implementation on top of an FPGA. The parameterized FPGA-based architecture has a global communication system based on multistage interconnection networks. The area cost scales with $O(n\log(n))$ and network delay scales with $O(\log(n))$ mapped onto FPGA technology. The placement and scheduling are simplified due to the global interconnection approach, as any FU could reach any FU. Moreover, the FUs can be homogeneous or heterogeneous depending on the architecture requirements. The SPR algorithm is a greedy heuristic with polynomial complexity which achieves results similar to related work and the CPU time is very small ranging from 10 to 300 milliseconds. The proposed approach could be included in just-in-time compilers or runtime environments.

We are currently using the SPR as a design exploration tool to evaluate a large set of architecture parameters and applications. A local register file in some FUs could reduce the initiation interval (II) achieved by the SPR algorithm, since in most cases the mapping fails due to register missing. We also intend to integrate the proposed SPR tool to compiler tools for softcore FPGA processors as MicroBlazer or NIOSII. The CGRA tightly-coupled with a softcore processor could increase significantly the performance of data intensive applications.

7. ACKNOWLEDGMENTS

This work is partially supported by the following Brazilian institutions, agencies and companies: Informatics Institute - Universidade Federal do Rio Grande do Sul (UFRGS) and Universidade Federal de Vicosa (UFV), Capes/DAAD/PROBRAL (Edital DRI/CGCI n. 014/2009 - Project 343/10), Fapemig (CEX APQ 00472-10), Fapergs, CNPq, Funarbe, Sydle, Gapso.

8. REFERENCES

[1] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. *Design, Automation and Test in Europe Conference and Exhibition*, 1:81, 2006.

[2] E. Benchmarks. Electrical & Computer Engineering Department at the UCSB, USA. http://express.ece.ucsb.edu/benchmark/, last access on 3rd february 2011.

[3] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[4] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *FPL '98: Proc. International Workshop on Field-Programmable Logic*, pages 213–222, 1998.

[5] S. Borkar. Electronics beyond nano-scale CMOS. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 807 –808, 2006.

[6] C. Clos. A study of non-blocking switch networks. Technical report, Bell System Tech. J. 32:407-425, March, 1953.

[7] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *IEEE/ACM CODES+ISSS*, pages 13 – 22, 2010.

[8] A. DeHon. Very large scale spatial computing. In *Unconventional Models of Computation*, volume 2509 of *Lecture Notes in Computer Science*, pages 27–37. Springer Berlin / Heidelberg, 2002.

[9] R. Ferreira, J. Vendramini, and M. Nacif. Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. In *IEEE 9th International Conference on Industrial Informatics*, INDIN, 2011.

[10] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 191–200, New York, NY, USA, 2009. ACM.

[11] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: a co-processor for streaming multimedia acceleration. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 28–39, Washington, DC, USA, 1999. IEEE Computer Society.

[12] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proc. conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[13] Y. Kim, I. Park, K. Choi, and Y. Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proceedings of the 2006 international symposium on Low power electronics and design*, ISLPED '06, pages 310–315, New York, NY, USA, 2006. ACM.

[14] G. Lee, S. Lee, K. Choi, and N. Dutt. Routing-aware application mapping considering steiner points for coarse-grained reconfigurable architecture. In P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 231–243. Springer Berlin / Heidelberg, 2010.

[15] L. McMurchie and C. Ebeling. Pathfinder: A negociated-based performance-driven router for FPGA. In *ACM/IEEE FPGA Conference*, pages 111–117, 1995.

[16] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10296, Washington, DC, USA, 2003. IEEE Computer Society.

[17] N. Mir. A survey of data multicast techniques, architectures, and algorithms. *Communications Magazine, IEEE*, 39(9):164 –170, Sept. 2001.

[18] B. Neji, Y. Aydi, R. Ben-atilallah, S. Meftaly, M. Abid, Dykeyser, and J-L. Multistage interconnection network for MPSoC: Performances study and prototyping on FPGA. In *IEEE Design and Test Workshop (IDT)*, 2008.

[19] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[20] C. Schack, W. Heenes, and R. Hoffmann. A multiprocessor architecture with an omega network for the massively parallel model gca. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 98–107. Springer Berlin / Heidelberg, 2009.

[21] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5), 2000.

[22] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[23] K. Tanigawa, T. Zuyama, T. Uchida, and T. Hironaka. Exploring compact design on high throughput coarse grained reconfigurable architectures. In *FPL '08: Proc. International Workshop on Field-Programmable Logic*, pages 126–135, London, UK, 2008.

[24] J. C. G. Vendramini and R. Ferreira. Parallel routing algorithm for extra level omega networks on reconfigurable systems. *WSCAD Symposium on Computing Systems*, 0:1–8, 2010.

[25] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 17:1565–1578, November 2009.