A Polynomial Placement Algorithm for Data Driven Coarse-Grained Reconfigurable Architectures

Ricardo Ferreira * Alisson Garcia † Tiago Teixeira
Depto de Informática - Universidade Federal de Vicosa
Vicosa - Cep 36570-000, Brazil
ricardo@ufv.br

João M. P. Cardoso
INESC-ID/IST/UTL
1000-029, Lisboa, Portugal
jcmp@acm.org

Abstract

Coarse-grained reconfigurable computing architectures vary widely in the number and characteristics of the processing elements (cells) and routing topologies used. In order to exploit several different topologies, a place and route framework, able to deal with such vast design exploration space, is of paramount importance. Bearing this in mind, this paper proposes a placement scheme able to target different topologies when considering data-driven reconfigurable architectures. Our approach uses graph models for the target architecture and for the dataflow representation of the application being mapped. Our placement algorithm is guided by a Depth-First Traversal in both the architecture and the application graphs. Two versions of the placement algorithm with respectively O(e) and O($e + n^3$) computational complexities are presented, where e is the number of edges in the dataflow representation of the application and n is the number of cells in the graph model of the architecture. The achieved experimental results show that our approach can be useful to exploit different interconnect topologies as far as coarse-grained reconfigurable computing architectures are concerned.

1. Introduction

Coarse-grained reconfigurable architectures have shown interesting achievements [5]. Those architectures may rely on very different computational models. For instance, since those architectures behaving in a static dataflow fashion [12, 6] naturally process data streams, they are seen as very promising solutions for the widely used stream-based computing applications. The granularity of a reconfigurable architecture is defined according to the bit-width of the operands of the functional units (FUs) used. Fine-grained reconfigurable architectures (e.g., FPGAs) have FUs with low bit-widths (e.g., 1-bit 4 inputs and 1-bit 1 output). Low granularity often implies a greater flexibility. However, there is a penalty associated with this in terms of power dissipation, area and delays, due to the higher number of routing resources required to map an application. Coarse-grained processing elements are better optimized for standard data-path applications. A large number of coarse-grained reconfigurable architectures have been presented in recent years [5], in order to overcome some of the limitations of the fine-grained structures used in common FPGAs. In fact, coarse-grained architectures are more suited to implement typical DSP algorithms, require shorter design cycles, and are faster to reconfigure, etc.

Many array architectures seem to be designed without strong evidences for the architectural decisions taken. Remarkably, the work presented in [5] has been one of the few exceptions which addressed the exploration of architectural features [6, 9]. With similar design exploration goals, especially to help the designer to systematically investigate different interconnect topologies, our approach presents a framework able to model and target different coarse-grained reconfigurable architectures. Since the design space exploration of coarse-grained reconfigurable architectures needs fast and flexible placement and routing algorithms, the main contribution of this work is a polynomial and flexible placement algorithm able to address different architectures. To validate our approach, a set of DSP kernels has been mapped on architectures with different interconnect topologies. The experimental results show the capability of new hybrid topologies to improve Grid plus 1-hop topologies as far as minimum path lengths and better performance are concerned.

This paper is organized as follows. Next section briefly introduces coarse-gained reconfigurable computing architectures and routing topologies. Section 3 explains two placement algorithms: with linear and cubic computational complexity, respectively. In section 4, experimental results are shown and new hybrid topologies are presented. The

*Finantial Support: FAPEMIG
†Finantial Support: Pibic/CNPQ

IEEE Computer Society Annual Symposium on VLSI(ISVLSI'07)
0-7695-2896-1/07 $20.00 © 2007 IEEE

related work is described in section 5. Finally, section 6 concludes this paper.

2. Coarse-Grained Reconfigurable Array Architectures

Coarse-grained reconfigurable array architectures, referred herein as array processors, can be defined as a set of cells connected with different topologies. Each cell is associated with one or more FUs responsible to perform a number of operations (including arithmetic, logic, and special operations to deal with conditional branches). Array processors are coupled to microprocessors and are usually seen as specific co-processors. They are used to globally accelerate applications by executing certain kernels for which they are more suited than traditional microprocessors. Array processors are programmed to implement computational structures using configurations. Each configuration defines the operations in the FUs of each cell and the interconnections among them. In certain architectures, each FU may execute more than one operation in the same configuration.

The topologies to interconnect the cells in the array can be very different. The differences reside on the number of interconnects between neighbors (adjacent cells) and non-adjacent cells, type of those interconnects (input/output or bidirectional), existence of buses, etc. Let us consider the topologies where each cell has eight neighbors. Figure 1 shows some examples of those topologies. They can be octal (Figure 1a), Grid (Figure 1b), or Grid plus 1-hop (Figure 1c), where each cell is connected to all the adjacent cells (0-hop) and to all the cells that can be reached throughout the immediate horizontal and vertical neighbor cells (1-hop). Concerning routing topologies for array processors, there is a large space for design exploration that, to the best of our knowledge, has not been sufficiently and effectively exploited. Known work has focused on limited templates or on a small number of routing topologies.

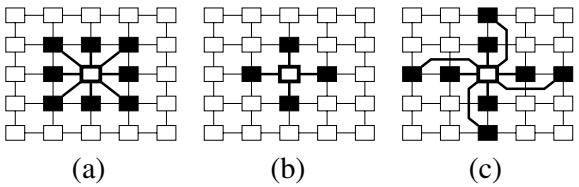

Figure 1. Topologies: (a) Octal; (b) Grid; (c) Grid plus 1-hop.

Let us consider an architecture graph model where each node is based on a cell interconnect-wrapper. This model can easily handle different interconnect topologies. The properties of the node define the internal architecture and routing capacity. A node may have only the routing function, behaving as a switch, or may include a computing function, as an FU. The interconnect resources between two wrappers are fixed and are defined by the array topology. Our approach focuses on design exploration of different local patterns between the interconnect-wrappers. Figure 2a displays an architecture with Grid topology. Figure 2b shows a configuration where the output of FU A is connected to the input of FU F routed by the path A-D-E-F. Figure 2b also shows the output of FU C directly connected to the input of FU B by using one local connection. The routing cost is defined by a function of the number of local interconnections needed to connect FUs. In Figure 2b, the path from A to F has cost three and from C to B has cost one. In this work, a placement algorithm addressing the minimization of the total routing cost is presented. Next section explains this algorithm.

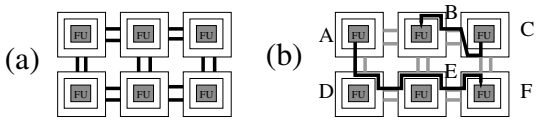

Figure 2. Fixed Local Connection: (a) Grid topology; (b) Two Routing Paths.

3. Placement Algorithms

The place and route is a well known NP-Complete problem. The large permutation space, e.g., for $N \times M$ array, the solution space is $(N \cdot M)!$, makes, even for small arrays, an exhaustive search prohibitive. Greedy, simulated annealing (SA) and genetic algorithms (GA) can reach good trade-offs between CPU time and interconnection wire costs. Previous work has shown that even for a random initial population in a 10×10 array architecture (i.e., 100! solution space), a genetic algorithm can reach a routable placement in few seconds [11]. However, faster placements might be needed to explore large design spaces and/or to perform dynamic placement. The work presented here presents two faster placement algorithms.

Since we have in mind early explorations of array architectures, our current approach uses a graph to model the architecture, instead of the less flexible, previously presented approach [11]. Such graph model will enable future work to address semi-automatic generation of array architectures that best suited a number of requirements.

3.1. Linear Algorithm

The placement problem is addressed here by considering a simultaneously traversing in depth first order both the architecture and the dataflow graphs. The pseudo-code of the algorithm is shown in Figure 3. For each path, during the dataflow graph traversal, the index of the node where the next path starts is stored in the last node of the current path. The function *PlaceDepthFirst* performs the placement of all the dataflow graph nodes in each path (stored in the *path* list) and returns the next initial cell for the next path.

Let us consider the dataflow graph for a 4-tap FIR shown in Figure 4a, where the number associated with each node represents the depth first order. The first path starts at node X and goes to node Y (i.e., 1 to 6). An initial cell of the architecture is chosen (w0, the top left cell in this example), and the nodes $X, \ldots, Y$ are placed in depth first traversal on the graph model of the architecture, see Figure 4d. The node Y stores the index 2 (the start node of the next path in the dataflow graph), which corresponds to node C1, and a child of the cell node where C1 is placed (cell w4) will be the initial cell of the architecture (e.g., cell w5) for the next placements. Then, the second path starts at node $C2$ and goes to node $A2$ (i.e., 7 to 10), as shown in Figure 4b and Figure 4e. Finally, the third (i.e. $M3$) and the fourth paths (i.e. $M2$) are placed, as is shown in Figure 4c and Figure 4f. If all the adjacent cells to the current cell position are occupied, the nearest free cell in the architecture graph is chosen to continue the traversal. The algorithm computational complexity is linear, O(e), where e is the number of edges in the dataflow graph.

```
initial_cell=w0; path = empty;
L = DepthFirst_EndMark(Dataflow_Graph);
While ( L.not_empty() ) {
  x = L.remove_first();
  path.add(x);
  if ( x.end_path() ) {
    initial_cell= PlaceDepthFirst(path,
                  initial_cell,x.get_index());
     path = empty;
  }
}
```

Figure 3. Linear Placement Algorithm.

3.2. Cubic Algorithm

We also propose here a O($e + n^3$) placement algorithm, being n the number of cells of the target architecture and e the number of edges in the dataflow graph. The pseudo-code of the algorithm is shown in Figure 5. First, a Dijsktra's algorithm is executed to find all the shortest paths between each cell pair in the architecture graph, and to create

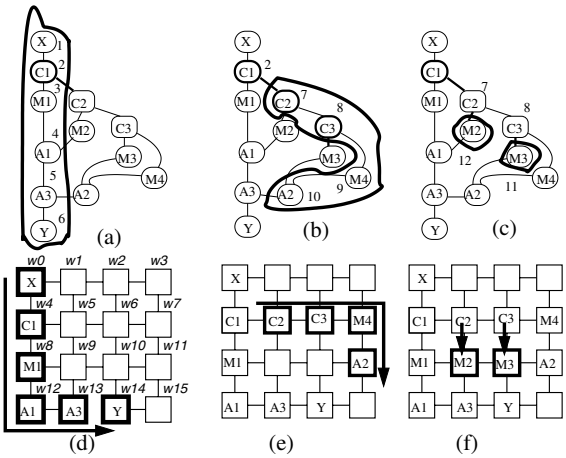

Figure 4. Dataflow graph and placement: (a), (d) First path; (b), (e) Second path; (c), (f) Last two paths.

```
Dijsktra_All_Cell(archicteture); // O(n3)
For i = 0 to K { // K is the number of tries
  S = Step_Random_LinearPlacement(dataflow,
                  architecture);
  c= routing_cost(S);
  if ( c < best_cost ) {
    Solution = S;  best_cost = c;
  }
}
Route(Solution);
```

Figure 5. Cubic Placement Algorithm.

a routing cost function. This algorithm is executed once for a given architecture, and has O(n^3) computational complexity. After this, placements can be done with O(e) computational complexity. A random depth first search is used to generate a set of placement solution. Let us consider again the dataflow graph shown in Figure 4a. During the architecture traversal, a cell child is randomly chosen. Figure 6 displays a different placement generated by another depth first traversal in the architecture graph. Then, the placement solution with a smallest estimated routing cost is chosen. The current routing algorithm is based on the Dijsktra's algorithm. Note, however, that the flexibility of our framework easily permit to add a pathfinder routing algorithm, for instance.

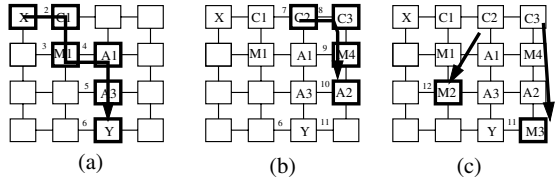

Figure 6. Placement steps: (a) First Path; (b) Second Path; (c) Overall Placement.

Table 1. Benchmark characteristics.

Benchmark	Graph Node	Graph Edge	Unit Type mul	Unit Type control	Unit Type alu	Unit Type i/o
fir16	46	60	16	15	16	2
fir64	193	255	64	63	64	2
Tree Fir64	193	255	64	63	64	2
Cplx8	46	60	8	22	14	2
FilterRGB	57	70	6	23	25	3
FilterRGB+Paeth	82	104	6	35	34	7
FilterRGB+ Paeth+FDCT	221	290	20	105	70	9
SNN	249	295	48	95	82	10
FDCT	139	186	14	70	52	2
FDCTa	92	124	14	40	36	2

4. Experimental Results

The two previously described placement algorithms have been used to evaluate the mapping of a set of benchmarks on different architectures. Those benchmarks include the following DSP algorithms: FIR, CPLX, FDCT, Paeth, FilterRGB and SNN. FIR is a 1-D finite-impulse response filter (versions with different number of taps are used). CPLX is a FIR filter using complex arithmetic. FDCT is a fast discrete cosine transform implementation. FDCTa is a part of the FDCT related to the vertical traversing. Paeth is the PNG (Portable Network Graphics) Paeth encoding routine. The Filter RGB is an image filter to highlight an image by brightening or darkening the pixels in the images. SNN is a symmetric nearest neighbor image filter. The input algorithms have been manually translated to a dataflow representation. This translation has been done considering optimization techniques that are typically used when compiling software programming language (e.g., C [3]) to data-driven representations.

Table 1 shows the main characteristics of the dataflow graph (number of nodes and edges) and the number of resources needed (number of FUs and operators) for each benchmark. Note that although some simple examples have been used, complex ones, as far as the mapping of kernels to a single configuration of an array processor is concerned, also have been tested (e.g., the FDCT example requires 139 data-path operations).

By placing dataflow graphs on different areas of an architecture, different tasks can be concurrently executed and/or the reconfiguration time of certain tasks can be hide with the execution of other tasks. Lines FilterRGB+Paeth and FilterRGB+Paeth+FDCT show the number of resources needed after performing place and route of multiple kernels.

Since previous experimental results (e.g., [1, 9]) suggested that the 1-hop topology is one of the most suited topologies for DSP benchmarks, it is used here to evaluate the performance of our algorithms. Table 2 shows the mapping results for all the considered benchmarks on 1-hop array topology by using our linear and cubic placement algorithms: O(e) and O($e + n^3$). Let us consider the linear placement O(e). Column **Avr** shows the average path lengths (measured as the number of cells that a connection needs to traverse from the source to the sink cell) after mapping the examples onto 1-hop topology. Columns C1 display the percentage of neighborhood connections (i.e., routing cost 1). Column C1_2 shows the percentage of routing connections with costs equal or less than 2. For instance, the Average path length for the mapping shown in Figure 4 has cost $=\frac{12\times1+1\times2+1\times3}{14} = 1.13$, C1 is equals to $12/14 * 100 = 85\%$, and C1_2 $=13/14 * 100 = 93\%$. The results show that most of routing connections are of neighborhood type. Let us consider the O($e + n^3$) placement algorithm. Columns 10 and 50 show the average path lengths for the best placement when the Depth First is performed 10 and 50 times, respectively. Note that, as aforementioned, each depth first step uses a random choice to select a child cell during the graph traversal. The O($e + n^3$) placement algorithm reduces the routing cost by a factor of 5-10%.

Table 2. Routing Costs for Grid 1-hop.

Benchmark	O(e) Avr	O(e) C1	O(e) C1_2	O(e+n^3) 10	O(e+n^3) 50
fir16	1.43	80%	93%	1.37	**1.28**
fir64	1.92	62%	74%	**1.80**	1.90
tree fir64	1.61	72%	85%	1.54	**1.47**
Cplx8	1.5	75%	85%	**1.35**	**1.35**
FilterRGB	1.31	80%	94%	**1.24**	1.29
FilterRGB+Paeth	1.47	73%	81%	**1.43**	**1.43**
FilterRGB+Paeth+FDCT	1.97	67%	78%	**1.86**	**1.86**
SNN	1.84	74%	84%	1.85	**1.83**
FDCT	1.91	67%	73%	1.86	**1.84**
FDCTa	1.91	68%	76%	1.75	**1.74**

Multi-stage networks have been studied for building parallel computers with hundreds of processors and have been used in some commercial parallel computers. The interconnect patterns are based on bitwise permutation and/or logic operators. Those properties are very useful to de-

velop algorithms for dynamic routing. For instance, the baseline pattern is computed by performing a shift right on n lower cell address bits. Let us consider the cell address $b_{n-1} \ldots b_i b_{i-1} \ldots b_1 b_0$. The baseline operator on bit i will generate the new address $b_{n-1} \ldots b_{i+1} b_0 b_i b_{i-1} \ldots b_1$, as shown in Figure 7a for a two column 8 cells network. Considering the address 3(011) in column 0 and the bit baseline b_2, the target connection in column 1 will be the address 5(101). Note also that other multi-stage patterns could be also supported by the approach presented in this paper (e.g., shuffle-exchange, inverse bit, and butterfly topologies).

A new hybrid local pattern can be created by mixing the baseline pattern with 1-hop patterns. We apply the standard 1-hop pattern to connect each PE to four neighbors, e.g., the $PE_{i,j}$ connects to $PE_{i,j+2}$, $PE_{i+2,j}$, $PE_{i,j-2}$, $PE_{i-2,j}$. Other four neighbors could be connected to each PE by applying the baseline operator on bit k in either column or line addresses, e.g., $PE_{i,j}$ connects to $PE_{baseline(i,k),j+1}$, $PE_{i+1,baseline(j,k)}$, $PE_{baseline(i,k),j-1}$, $PE_{i-1,baseline(j,k)}$. Figure 7b shows the baseline 1-hop pattern, where $PE_{3,j}$ has an 1-hop connection to $PE_{3,j+2}$ and a 0-hop plus baseline on bit 2 to $PE_{5,j+1}$. An architecture with a baseline plus 1-hop interconnect pattern is considered during the evaluation of the proposed placement algorithms.

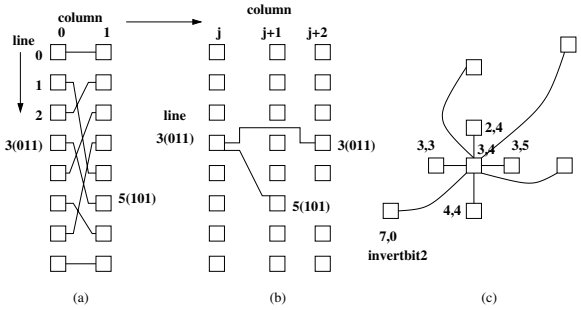

Figure 7. Multi-stage topologies: (a) Baseline with 3 bit address; (b) Baseline plus 1-hop; (c) Cube plus 0-hop.

Multi-degree networks, such as hypercube and k-ary n-cube, can be also mixed with 0-hop interconnects. A local pattern can be generated by using four 0-hop interconnections in horizontal and vertical direction, plus cube interconnections, each one obtained by switching a bit in line and/or column addresses. For instance, $PE_{3,4}$ has a 0-hop interconnection to $PE_{2,4}$, $PE_{4,4}$, $PE_{3,5}$ and $PE_{3,3}$ (see Figure 7c), and $PE_{3(\underline{0}11),4(\underline{1}00)}$ has a cube interconnection to $PE_{7(\underline{1}11),0(\underline{0}00)}$. In the context of this paper, an architecture with a cube plus 0-hop interconnect pattern is also considered when evaluating the proposed placement algorithms.

We show experiments by bounding each cells connectivity to eight neighbors and a Grid architecture with 1-hop topology is used as reference. Table 3 summarizes the relative improvements when comparing each exploited topologies (octal 0-hop, baseline plus 1-hop, and cube plus 0-hop) with grid 1-hop. Columns A and B show the routing improvements when the $O(e)$ and $O(e + n^3)$ placement algorithms, explained in this paper, are applied, respectively. The results achieved with the $O(e + n^3)$ placement algorithm have been obtained by performing 10 random depth first searches. The $O(e+n^3)$ placement algorithm achieved always better results than the $O(e)$ algorithm. Note, however, that the results achieved with the linear placement algorithm are only about 6% worse on average than the ones obtained by the cubic placement algorithm. Those results also indicate the better suitability of the cube plus 0-hop topology over grid with 1-hop, octal, and baseline plus 1-hop topologies.

Table 3. Routing cost improvements over Grid 1-hop.

	octal		baseline+1hop		0hop+cube	
Benchmark	A	B	A	B	A	B
fir16	-23%	-10%	1%	2%	**4%**	**4%**
fir64	-38%	-22%	-3%	15%	-3%	**10%**
tree fir64	-30%	-23%	1%	7%	-3%	0%
Cplx8	-10%	4%	1%	9%	9%	**14%**
FilterRGB	-16%	-10%	0%	3%	4%	**7%**
FilterRGB+Paeth	-14%	-5%	-2%	6%	0%	**8%**
FilterRGB +Paeth+FDCT	-13%	-7%	16%	**18%**	12%	**18%**
SNN	-19%	-18%	8%	10%	16%	**18%**
FDCT	-20%	16%	2%	5%	18%	**20%**
FDCTa	-16%	-8%	9%	16%	19%	**25%**

5. Related Work

Previous work has addressed the mapping of data-driven algorithms on regular array architectures [10, 1, 2, 8, 9]. A place and route algorithm for a regular array architecture has been proposed in [7]. In [10], a simulated annealing approach is presented to generate a more compact array. Both approaches analyze only the hexagonal topology with a fixed interconnect degree. On the other hand, most of array architectures are based on grid topologies [9, 1, 2, 8]. In [2], cluster architectures are addressed. However, the authors [2] focus only on dynamic resources allocation over small heterogeneous arrays (e.g., arrays with 12 to 16 cells), and their hierarchical communication costs. Recently, Bansal et al. [1] have explored a number of interconnect patterns such as grid, 1-hop and 2-hop.

Most placement strategies use heuristics or simulated an-

nealing based algorithms. Recently, Lai et al. [8] presented a placement algorithm with computational complexity proportional to the square number of the nodes in the DFG. However, the algorithm is too specific to the rDPA architecture (cells are neighborly connected using 2-inputs/2-ouputs patterns) to be used in a highly flexible environment.

In [9], an exploration approach based on a flexible environment using C, XML and a retargetable simulator is presented. The cell allocation uses a modulo scheduling algorithm. The array design exploration is limited to grid, 1-hop and MorphoSys architectures. The results suggest that the 1-hop pattern produces the same performance of MorphoSys, but requires considerably less silicon area and configuration bits.

In [4], a genetic algorithm to perform placement on coarse-grained reconfigurable architectures is presented. The properties of the target architecture have made possible to integrate the routing in the genetic algorithm, but it does neither address routing congestion nor scalability. Moreover, the experimental results address a specific architecture and small kernels are used.

Our approach is different from the previous ones in a number of aspects. As far as coarse-grained, data-driven, array architectures are targeted, the most similar work is the one related to the Lai et al. [8] polynomial placement. However, our placement is not constrained by a specific array topology (e.g., mesh-based arrays) and is, therefore, able to support a wide range of data-driven arrays with different connections and array topologies.

6. Conclusion

This paper presents two placement algorithms for data-driven, reconfigurable, coarse-grained architectures. The placement algorithms have linear and cubic computational complexity, respectively. Those algorithms and the flexible architecture graph model have been developed to easily evaluate different routing topologies. Particularly, mapping results achieved by evaluating a number of architectures with different routing topologies (e.g., 1-hop, 1-hop plus multi-stage patterns, and n-cube) have been shown. Those results strongly expose the capability of the baseline 2/1-hop, and 0-hop/cube to improve 1-hop topologies concerning minimum path lengths and performance. The placement algorithm, combined with the graph model of the architecture, is not limited to the routing topologies presented through this paper. Thus, our approach can be used to exploit reconfigurable architectures considering high-degrees of freedom as far as interconnect topologies are concerned.

Short term plans include the automatic generation and exploration of coarse-grained architectures in order to propose suitable routing topologies for a certain set of benchmarks. We also plan to test simulated annealing and greedy approaches to improve the depth first order traversal in the architecture graph model. Those approaches may exploit even better the locality existent in the dataflow graph of the applications being mapped.

References

[1] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta. Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10474, Washington, DC, USA, 2004. IEEE Computer Society.

[2] L. Bossuet, G. Gogniat, and J. Philippe. Fast Design Space Exploration Method for Reconfigurable Architectures. In *The International Conference on Engineering of Reconfigurable Systems and Algorithm, ERSA'03*, Las Vegas, Nevada, USA, June 23-26 2003.

[3] M. Budiu and S. C. Goldstein. Compiling Application-Specific Hardware. In Springer-Verlag, editor, *Proceedings 12th Int'l Conference on Field Programmable Logic and Applications (FPL'02)*, volume LNCS 2438, pages 853–863, 2002.

[4] C. P. Fred Ma, John Knight. Physical resource binding for coarse-grain reconfigurable array using evolutionary algorithms. *IEEE Transctions on Very Large Integration (VLSI)*, 13(5), May 2005.

[5] R. Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Int'l Conf. on Design, Automation and Test in Europe (DATE'01)*, pages pp. 642–649, March 12-15 2001.

[6] R. Hartenstein, R. Kress, and H. Reinig. A Dynamically Reconfigurable Wavefront Array Architecture for Evaluation of Expressions. In *Application Specific Array Processors, 1994. Proceedings., International Conference on*, pages 404–414, Aug. 22-24 1994.

[7] I. Koren, B. Mendelson, I. Peled, and G. M. Silberman. A data-driven VLSI array for arbitrary algorithms. *Computer*, 21(10):30–43, Oct. 1988.

[8] Y.-T. Lai, H.-Y. Lai, and C.-N. Yeh. Placement for the reconfigurable datapath architecture. In *IEEE International Symposium on Circuits and Systems, ISCAS*, volume 2, pages 1875 – 1878, 2005.

[9] B. Mei, A. Lambrechts, D. Verkest, J.-Y. Mignolet, and R. Lauwereins. "architecture exploration for a reconfigurable architecture template". *IEEE Design and Test of Computers*, 22(2):90–101, Mar/Apr 2005.

[10] B. Robic and B. Vilfan. Improved schemes for mapping arbitrary algorithms onto processor meshes. Technical Report CSD-TR-95-3, Computer Systems Department, Jozef Stefan Institute, Mar. 15 1995. Sat, 22 Jun 1996 10:22:05 GMT.

[11] V. Silva, R. Ferreira, A. Garcia, and J. Cardoso. Mesh mapping exploration for coarse-grained reconfigurable array architectures. In *Proceedings of 3rd International Conference on ReConFigurable Computing and FPGAs 2006, ReConFig06*, San Luis Potosi, Mexico, September 2006.

[12] A. Veen. Dataflow Machine Archirecture. *ACM Computing Surveys*, 18:365–396, 1986.