OPTIMAL SIMULTANEOUS SCHEDULING, BINDING AND ROUTING FOR PROCESSOR-LIKE RECONFIGURABLE ARCHITECTURES

J. A. Brenner, J. C. van der Veen, S. P. Fekete

Department for Mathematical Optimization
Braunschweig University of Technology
Braunschweig, Germany
{j.brenner, j.van-der-veen, s.fekete}@tu-bs.de

J. Oliveira Filho, W. Rosenstiel

Department for Computer Engineering
University of Tübingen
Tübingen, Germany
oliveira@informatik.uni-tuebingen.de
rosenstiel@informatik.uni-tuebingen.de

ABSTRACT

We discuss the problem of *simultaneously* scheduling, binding and routing a given data flow graph to a coarse-grain architecture consisting of identical processing elements (PEs) that are connected by a nearest-neighbour mesh-like interconnection network.

While there are heuristics trying to solve this problem, we develop the first *exact* method based on integer linear programming. This allows us to achieve provably optimal solutions for two different objective functions, for small to medium instances. In addition, we describe a heuristic that seems to outperform all other known heuristics.

1. INTRODUCTION

In the past decade, many coarse-grain reconfigurable architectures have been proposed both by industry as well as by academia [1]. In this paper we focus on a subclass: architectures that decompose into a set of identical processing elements (PEs), connected by a mesh-like communication infrastructure called *coarse-grain reconfigurable arrays* (CGRA) (see for example [2, 3, 4, 5]).

In this paper we tackle the following fundamental problem: Given an application in form of a data-flow graph, schedule all operations as to minimize the total execution time while simultaneously determining an assignment of the operations to the PEs (binding) and shortest conflict free routes (routing) for the data. Remarkable is that this optimization process requires a combination of steps (scheduling, binding, routing) that also show up in other areas of VLSI. Traditionally, these aspects were dealt with sequentially; our approach is to deal with all of them at once, leading to better results.

Supported by DFG Priority Program 1148 on Reconfigurable Computing Systems, Grants FE 407/8-1, FE 407/8-2, RO 1030/13-1, and RO 1030/13-2 and by Programme Alßan, European Union Programme of High Level Scholarships for Latin America, n. E04D045457BR.

There are a number of previous approaches that tackle the problem heuristically. Bansal et al. [6], present a priority-based list scheduling heuristic; they also hint at an integer linear programming (ILP) formulation, without getting publishable results. Dimitroulakos et al. [7] consider a variation, with the main focus being memory bandwidth.

The main contribution of this paper is an ILP formulation for the problem of simultaneous scheduling, binding and routing (SSBR). As we will demonstrate, this allows using state-of-the-art ILP solvers to solve small- to medium-sized instances to optimality in less than an hour. Our exact procedure is augmented by a heuristic that is capable of solving large instances in very short time.

The rest of the paper is organized as follows. In the next section we give a short description of the configurable reconfigurable core (CRC) architecture. In Section 3 we describe our integer linear programming formulation. A new heuristic is described in Section 4. Both ILP and heuristic have been benchmarked on a set of instances. The computational results are given in Section 5. Concluding remarks are given in Section 6.

2. CRC ARCHITECTURE

The CRC is a parameterizable model for a processor-like reconfigurable architecture. The term *processor-like reconfigurability* [8] alludes to the ability of the architecture to perform reconfiguration as part of the regular execution pace. As opposed to FPGAs, whose reconfiguration phase takes several clock-cycles, processor-like reconfiguration allows instantiation and execution of exactly that part of a circuit that is needed in the current clock-cycle. The CRC architecture has been shown to increase performance when targeted to an application domain such as computer vision [9], and to introduce new power optimization possibilities [10].

One possible instance of the CRC model, developed at the University of Tübingen, is depicted in Figure 1. It

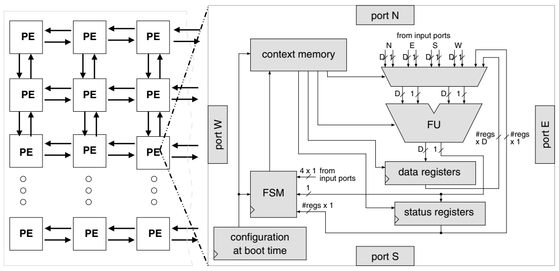

Fig. 1. The CRC Architecture: An interconnection network implements a nearest-neighbor variant; the zoom shows the basic PE structure in detail.

consists of a two-dimensional array of processing elements (PEs), surrounded by an interconnection network. The array geometry, i.e., its length and width, and the type of interconnection network are parameters of the model. In the following, we consider a nearest-neighbor network variant. It means each PE may only transfer and receive data to/from horizontally or vertically adjacent PEs. A PE may also be used as a routing element. In that case, it preserves its ability to execute one operation. The PEs on the border of the array provide access to ports and memory modules.

Each PE is composed of a finite-state machine module and a data path. Extremely fast reconfiguration (within one clock-cycle) is obtained by providing each PE with a context memory. In the beginning of a cycle, the finite-state machine reconfigures the PE by selecting one entry of the context memory. Each entry provides information that will determine which operation is to be executed in the functional unit, where the operands come from, and where the result of the operation is to be stored. This CRC instance admits as operations all C operators, excluding / and $\%$. Operands may come from any neighbor or from the internal PE register set. Similarly, the result of an operation may be stored in the internal register set or routed to another PE.

During the execution phase, each PE will use input data produced in previous cycles, possibly by other PEs. It will also produce new data that should be routed to the points where it is needed. In order to take advantage of fast reconfigurability, a proper temporal and spatial partitioning of the application must be found. Operations should be bound to PEs that are close to the origin of their input data, as transferring data is quite power-consuming. Consequently, a low-cost routing is extremely important.

3. ILP FORMULATION

The SSBR problem is NP-complete even for very simple types of data-flow graphs [11]; as a consequence, only heuristic solution methods have been published. In this section, we describe our ILP-based *exact* algorithm; to the best of our knowledge, it is the first algorithm for the SSBR problem that yields provably optimal results in reasonable time, for instances up to about 100 tasks. These optimal solutions serve as benchmarks for the development of superior heuristics (see Section 4).

Our formulation considers two different objective functions: the first aims at minimizing the number of clock-cycles needed to process a given data-flow graph; the other tries to minimize the storage times of intermediate values, given an upper bound on the total number of clock-cycles. Thus, solutions to this second ILP are more energy-efficient, because less data has to be transferred between PEs.

As we have stated above, we expect that the application to be executed is given as a directed acyclic data-flow graph $G = (V, A)$, unrolling loops if necessary.

There are three more parameters that strongly influence size and complexity of the ILPs. The most important one is *an upper bound on the number of clock-cycles*, $\overline{C}$. One way of computing this bound is the heuristic presented in Section 4; the closer this value is to the optimum, the smaller the resulting ILP; in general, the size of the ILP is positively correlated to the time needed for solving it. The other two parameters are the *architecture dimensions* $N \times M$. These parameters also influence the ILP size; note that they can have an effect on the solution value. When decreasing N and M, the optimal number of clock-cycles will increase, until there is no more feasible solution to SSBR.

Finally, each PE can communicate to its four axis-parallel neighbors. We allow each PE to either store or compute one new intermediate result per clock-cycle.

In the following we provide details of our formulation. This is done in three steps: first we introduce the variables used in the model (Section 3.1); then we list all of the constraints (Section 3.2); finally, we describe the two objective functions (Section 3.3).

3.1. Variables

We employ four different kinds of binary variables: *operation*, *storage*, *auxiliary storage*, and *usage* variables. *Operation* variables x_{isnm} specify when and where an operation is executed. Setting x_{isnm} to one indicates that operation i is executed on the PE with indices n and m in clock-cycle s. *Storage* variables y_{isnm} indicate that the result of an operation is stored for later use. A value of one for y_{isnm} means that in clock-cycle s the result of operation i is stored on PE (n, m); *auxiliary storage* variables z_{asnm} allow for conjoint routing of results to more than one target. If z_{asnm} is equal to one, at clock-cycle s the result of operation i is stored for operation j on PE (n, m). Here i and j are the source and target of edge $a = a_{ij} \in A$. We introduce *usage* variables u_s for technical reasons only. A variable u_s is set to one if at least one PE is used in step s.

The scheduling part of our ILP formulation is based on [12], which proposes a preprocessing algorithm for reducing the number of variables. This is based on as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling algorithms. As a consequence, operation variables x_{isnm} have indices $s \in S_i := \{1 + \text{ASAP}(i), \dots, \overline{C} - \text{ALAP}(i)\}$, storage variables y_{isnm} have indices $s \in S_i' := \{2 + \text{ASAP}(i), \dots, \overline{C} - 1\}$ and auxiliary storage variables $z_{a_{ij}snm}$ have indices $s \in S_{a_{ij}} := \{2 + \text{ASAP}(i), \dots, \overline{C} - 1 - \text{ALAP}(j)\}$, instead of $s \in \{1, \dots, \overline{C}\}$.

3.2. Constraints

Each operation has to be executed exactly once. This is expressed by the following *assignment constraints*:

$$\sum_{s \in S_i} \sum_{n=1}^{N} \sum_{m=1}^{M} x_{isnm} = 1 \qquad \forall\, i = 1, \dots, |V|. \quad (1)$$

The edges of the graph describe predecessor-successor relationships. They are reflected in *timing constraints*, $\forall\, a_{ij} \in A(G) \quad \forall\, s \in S_i \cup S_j$

$$\sum_{t \in S_i}^{t \geq s} \sum_{n=1}^{N} \sum_{m=1}^{M} x_{itnm} + \sum_{t \in S_j}^{t \leq s} \sum_{n=1}^{N} \sum_{m=1}^{M} x_{jtnm} \leq 1. \quad (2)$$

If for an edge a_{ij}, operation j is to be executed before or in clock-cycle s, operation i is a predecessor of j and must not be scheduled in cycle s or later.

Constraints (1) and (2) are taken from [12]; the polyhedron described by these two constraints is integral. This is no longer the case after more constraints have been added.

The *resource constraints* enforce that each PE can only execute or store one operation at a time: $\forall\, s \in S \quad \forall\, n = 1, \dots, N \quad \forall\, m = 1, \dots, M$

$$\sum_{i=1}^{numOps} \left(1_{s \in S_i} \cdot x_{isnm} + 1_{s \in S_i'} \cdot y_{isnm}\right) \leq 1. \quad (3)$$

Here, "1_{cond}" equals 1 iff the condition *cond* is true.

There are three types of *storing constraints*. If for an arc a_{ij}, j is not scheduled directly after i, the result of operation i has to be stored. This is indicated by a value of one for the corresponding variable $z_{a_{ij}snm}$. Let

$$s_k := \sum_{s \in S_k} \sum_{n=1}^{N} \sum_{m=1}^{M} s \cdot x_{ksnm}$$

denote the clock-cycle in which operation k is executed. Then for an edge a_{ij} exactly $s_j - s_i - 1$ of the z-variables need to be set to one. This is expressed by

$$s_j - s_i - \sum_{s \in S_a} \sum_{n=1}^{N} \sum_{m=1}^{M} z_{asnm} = 1 \quad \forall\, a_{ij} \in A(G). \quad (4)$$

An auxiliary storage variable $z_{a_{ij}snm}$ may only be set to one if operation i has been processed before clock-cycle s, so $\forall\, a_{ij} \in A(G) \quad \forall\, s \in S_a$

$$\sum_{n=1}^{N} \sum_{m=1}^{M} z_{a_{ij}snm} \leq \sum_{t \in S_i}^{t<s} \sum_{n=1}^{N} \sum_{m=1}^{M} x_{itnm}. \quad (5)$$

The right side of (5) is at most one, so the same intermediate result of operation i for the same operation j at any given clock-cycle s can exist only once. In order to allow the same result as an input to different operations, the auxiliary storing variables are linked to the storing variables as follows: $\forall\, s \in S_{a_{ij}}\ \forall\, n = 1, \dots, N\ \forall\, m = 1, \dots, M\ \forall\, a_{ij} \in A(G)$

$$y_{isnm} \geq z_{asnm}. \quad (6)$$

This allows both conjoint routing for multiple purposes and "split" storing while routing the same result to different PEs.

The above constraints deal with the PEs of the CRC architecture. Now consider the communication network; processing elements can only communicate with PEs that are axis-parallel neighbors. Thus, the Manhattan distance of two operations i and j connected by an edge (i, j) and bound to PEs (n_i, m_i) and (n_j, m_j) may not exceed one for direct communication, so

$$|n_j - n_i| + |m_j - m_i| \leq 1. \quad (7)$$

Otherwise, the result of i needs to be stored and routed to j in the next clock-cycles. If a result is stored and then used by another operation, there are three additional combinations for the relative location of two successive assignments:

(i) between x_{isnm} and the corresponding $z_{a(s+1)nm}$;

(ii) between consecutive storings z_{asnm} and $z_{a(s+1)nm}$;

(iii) and between the last storage z_{asnm} and $y_{j(s+1)nm}$.

These combinations are dealt with by the following *location constraints*, $\forall\, a_{ij} \in A(G) \quad \forall\, s = 1, \dots, \overline{C} - 1$:

$$\begin{aligned} &\left| \sum_{n=1}^{N} \sum_{m=1}^{M} n \cdot \left(x_{j(s+1)nm} + z_{a(s+1)nm}\right) - \sum_{n=1}^{N} \sum_{m=1}^{M} n \cdot \left(x_{isnm} + z_{asnm}\right) \right| \\ &+ \left| \sum_{n=1}^{N} \sum_{m=1}^{M} m \cdot \left(x_{j(s+1)nm} + z_{a(s+1)nm}\right) - \sum_{n=1}^{N} \sum_{m=1}^{M} m \cdot \left(x_{isnm} + z_{asnm}\right) \right| \leq 1 \end{aligned} \quad (8)$$

Note that the variables need only be included for those time steps in which they are defined; we omitted the "1_{cond}"-expressions for better readability.

Constraint (8) is nonlinear; this can be fixed by replacing each constraint $|a - b| + |c - d| \leq 1$ by the four expressions $(a-b)+(c-d) \leq 1$, $(a-b)+(d-c) \leq 1$, $(b-a)+(c-d) \leq 1$ and $(b - a) + (d - c) \leq 1$.

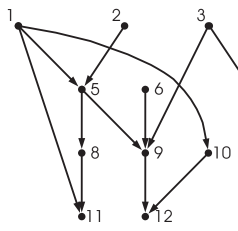

Fig. 2. A data-flow graph G.

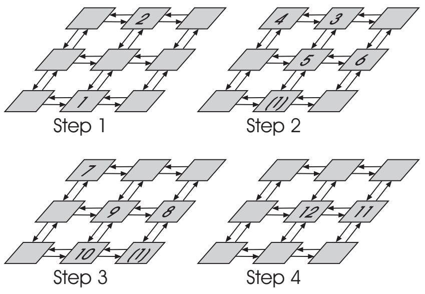

Fig. 3. The ILP embedding of G with storage minimization and $\overline{C} = 5, N = M = 3$. Numbers in brackets indicate that an operation is stored.

If some PE is used in clock-cycle s, we need to set the corresponding usage variable; this yields the *usage constraints* $\forall\, n = 1, \ldots, N\ \forall\, m = 1, \ldots, M,\ \forall\, i = 1, \ldots, |V|$

$$u_s \geq x_{isnm} \quad \forall\, s \in S_i \tag{9}$$

$$u_s \geq y_{isnm} \quad \forall\, s \in S'_i \tag{10}$$

The polyhedron described by constraints (1) through (10) contains all feasible solutions to the simultaneous scheduling, binding and routing problem. In the next section we add two objective functions to either minimize the number of clock-cycles or the number of stored intermediate results to completely process a given data-flow graph.

3.3. Objective Functions

We now give two alternative objective functions for the polyhedron of all feasible solutions to the SSBR problem.

The first aims at minimizing execution time; this can be done by minimizing the sum of the usage variables u_s:

$$\min \sum_{s \in S} u_s. \tag{11}$$

There are many scenarios where execution time is secondary to energy consumption, as the latter corresponds to the routing effort. Given a maximal number of clock-cycles in our model, this objective can be expressed by

Algorithm 1: Our heuristic algorithm.

```
Input: Digraph $G = (V, A)$, grid size $N \times M$
Output: Feasible binding $B : V \to \mathbb{N}^3, v \mapsto (n, m, s)$ and routing
        if successful

1  Initialize $s \leftarrow 1$
2  while availOpLeft do
3      OpAvail ← available operations
4      PEAvail ← all PEs
5      foreach $Op \in OpAvail$ do
6          foreach $PE \in PEAvail$ do
7              if routable and no unmovable label then
8                  Bind Op to PE in step $s$
9                  Remove labels for predecessors that are done
10                 If Op has out-edges, label PE for Op in next step
11                 Erase PE from PEAvail
12                 break for

13         if not bound then
14             $C \leftarrow$ FindCentralPE(Pred(Op))
15             foreach $p \in Pred(Op)$ do
16                 Move result of p towards C (may move labels)
17                 Mark used PE for p in next step
18                 Erase used PE from PEAvail

19     forall labels do
20         Bind labeled operation
21         Mark PE in next step
22     $s++$
```

$$\min \sum_{i=1}^{|V|} \sum_{s \in S'_i} \sum_{n=1}^{N} \sum_{m=1}^{M} y_{isnm}. \tag{12}$$

Before evaluating the ILP approach in Section 5, the following section is dedicated to a heuristic algorithm.

4. HEURISTIC

Now we describe our heuristic for the SSRB problem; it was primarily designed for providing an upper bound on the number of clock-cycles to be used as an input parameter for the ILP formulation. Even by itself, it computes near-optimal solutions in very short time.

Our heuristic is similar to the algorithm proposed by Bansal et al. [6] in that we assign priorities to PEs and operations and do clock-cycle binding. The approach in [6] seems to require an exponential-time subroutine called `IsRoutable` to integrate routing in the scheduling and binding part of their heuristic. In our approach, the routing of intermediate results is included in the step-wise procedure. Thus, no routings over more than one step need to be computed. The pseudo-code of our heuristic is given below.

The algorithm handles one clock-cycle in each iteration of a `while` loop. This loop terminates when all operations have been mapped or a deadlock condition is discovered. First, *OpAvail* is filled with all operations that are available in the current step s, i.e., all operations whose predecessors have already been scheduled. To increase the performance, operations with in-degree 0 whose successors will not be available in the next step are removed from *OpAvail*. As

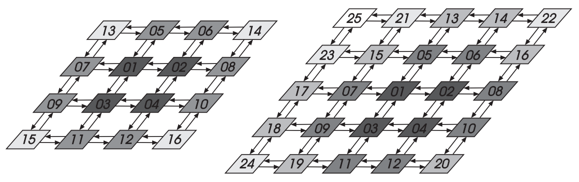

Fig. 4. PE priorities for the 4×4 and 5×5 grids – same shadings mean same priorities

operation priorities, we first consider the in-degree of a vertex, giving higher priority to vertices with many predecessors. In case of equality, the number of nodes on the longest path from *Op* is taken into account. Next, *PEAvail* is filled with all $N \times M$ PEs. We have compared three basic PE orderings: the "line-up" order sorts the PEs according to rows and columns, so that $prio(n, m) = n + m \cdot N$. The "snake" order proposed in [6] starts with a central PE and then rotates around it. However, the priority that turns out to be best measures the distance to the closest architecture corner, preferring PEs in central positions: $prio(n, m) = \min\{n, \overline{N} - n\} + \min\{m, \overline{M} - m\}$, where $\overline{N}$ and $\overline{M}$ denote the smallest even integers greater or equal to N and M, respectively. Figure 4 illustrates the priorities for two example grids; PEs with the same priorities are colored in the same shade of gray.

In order to avoid blocking, we have developed a special *labeling* system. Whenever the algorithm decides that an operation is executed or stored on a certain PE, we check if the operation has any successors that have yet to be scheduled; in that case we label the respective PE in the following time step, forcing the heuristic to reserve a reachable PE. However, we do not want labels to block a certain PE for more than one time step, so we allow moving them within the region reachable from their initial position. Marks are removed as soon as all successors of the respective operation have been bound, or at the end of the following iteration when the result is fixed to a PE.

The rest of the `while` loop can be divided into three parts: the first two deal with the available operations in the order of their priorities, the third buffers results needed for operations that are not yet available.

In the first part we iterate through all relevant PEs to find a feasible embedding for the current operation i (ll. 6–12). In doing so, preference is given to PEs with high affinities, which are computed for each individual operation i as follows. For each previously mapped predecessor j of a successor of i, we add an affinity point to every PE that lies at a distance of less than or equal to two from the location of j. Only in case of equal affinity, general PE priority is used.

If no feasible PE can be found, we enter the second part, in which the predecessors of operation i are moved to PEs that are as close to each other as possible (ll. 13–18). This is done to work towards a possibility of binding the operation in a subsequent clock-cycle.

The last part of the algorithm (ll. 19–21) handles leftover labels. These labels are placeholders for results required for the processing of operations that become available at a later time. The results are promptly fixed to PEs in the current clock-cycle. Instead of storing a result on the labeled PE, we compute its affinities to the reachable PEs as above and prefer PEs with higher affinity.

5. COMPUTATIONAL RESULTS

Now we demonstrate the usefulness of our approaches for solving the simultaneous scheduling, binding and routing problem. All computational experiments were carried out on an AMD Athlon64 X2 3800+, equipped with 2GB RAM, running under Linux. We used ILOG CPLEX 10.0 as the ILP solver, selecting eleven different benchmark instances from our code library. We tried hard but could not obtain the instances used in [6].

As the number of steps computed by the heuristic is used as an input parameter $\overline{C}$ of the ILP, we first discuss the results of our heuristic, as listed in Table 1 for the different instances. The first three columns give instance names, number of operations, and number of arcs of the data flow graph. The next columns give architecture dimensions, running time, and resulting number of clock-cycles. All in all, running times are negligible. For the ellip and xor4 instances, the heuristic cannot come up with a solution for too small dimensions.

In Table 2 we give the results for our ILPs. The first column shows instance names, with dimensions of the data flow graphs being the same as in Table 1. The second column lists the architecture dimensions. The third column shows the upper bound on the number of clock cycles. The column labeled "time[sec]" gives the time for solving the ILP for the objective function shown in the previous column; for small- to medium-sized instances, running times are tolerable. In the last column the optimal solution value as determined by the ILP is given for the objective functions (11) and (12). All other instances listed in Table 1 are not solvable in a time limit of 30 minutes.

6. CONCLUSION

In this paper we have presented the first exact method for the problem of simultaneously scheduling, binding, and routing a given data-flow graph for a coarse-grain reconfigurable array, such as the CRC architecture developed at the University of Tübingen. We demonstrate that state-of-the-art ILP solvers are able to solve this problem in reasonable time for small- and medium-sized instances. For larger instances we have developed a new heuristic that is able to compute near-optimal solutions in negligible time.

instance	$\|V\|$	$\|A\|$	$N \times M$	time[sec]	# steps
and	13	16	3×3	0.001	5
			4×4	0.002	5
			5×5	0.001	5
hist	14	13	3×3	0.001	7
			4×4	0.000	7
			5×5	0.003	7
xor	31	30	3×3	0.003	7
			4×4	0.001	7
			5×5	0.002	7
xor2	62	60	3×3	0.002	11
			4×4	0.000	8
			5×5	0.000	8
xor3	93	90	3×3	0.003	21
			4×4	0.002	12
			5×5	0.003	13
ellip	66	84	4×4	–	deadlock
			5×5	0.003	19
			6×6	0.004	18
sconv	85	84	3×3	0.002	18
			4×4	0.005	14
			5×5	0.000	16
xor4	124	120	3×3	–	deadlock
			4×4	0.001	19
			5×5	0.000	13
sconv2	172	170	4×4	0.004	23
			5×5	0.004	29
			9×9	0.003	27
sconv3	258	255	4×4	0.011	48
			5×5	0.004	37
			9×9	0.005	34
			20×20	0.007	39
fmm	704	688	9×9	0.469	54
			12×12	0.033	60
			20×20	0.093	44

Table 1. Results obtained by the heuristic for different benchmark instances and architecture dimensions.

The CRC is a parameterizable architecture model and also supports operator-chaining, multi-cycle operators and pipelining. We are optimistic to extend the ILP and the heuristic to adapt to architecture changes and yet unsupported functionality.

instance	N× M	$\overline{C}$	min	time [sec]	OPT
and	3× 3	5	u	0.04	5
	4× 4	5	u	0.09	5
	5× 5	5	u	0.24	5
	3× 3	5	y	0.07	1
	4× 4	5	y	0.10	1
	5× 5	5	y	0.42	1
hist	3× 3	7	u	0.07	7
	4× 4	7	u	0.19	7
	5× 5	7	u	0.67	7
	3× 3	7	y	0.06	0
	4× 4	7	y	0.10	0
	5× 5	7	y	0.18	0
xor	3× 3	7	u	1.06	7
	4× 4	7	u	1.50	7
	5× 5	7	u	2.09	7
	3× 3	7	y	0.44	0
	4× 4	7	y	0.30	0
	5× 5	7	y	1.51	0
xor2	3× 3	11	u	>1800.00	≥ 9
	4× 4	8	u	85.51	7
	5× 5	8	u	220.40	7
	3× 3	11	y	>1800.00	≥ 0
	4× 4	7	y	27.02	0
	5× 5	7	y	39.61	0
sconv	3× 3	18	u	378.51	14
	4× 4	14	u	2.62	14
	5× 5	16	u	>1800.00	≥ 14
	3× 3	14	y	1705.43	0
	4× 4	14	y	452.29	0
	5× 5	16	y	872.05	0

Table 2. Results obtained by the ILP for two different objective functions for different benchmark instances and architecture dimensions.

7. REFERENCES

[1] R. W. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective." in *DATE*, 2001, pp. 642–649.

[2] Elixent Ltd., http://www.elixent.com.

[3] M. Motomura, "A dynamically reconfigurable processor architecture," in *Microprocessor Forum*, October 2002.

[4] V. Baumgarten, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "Pact xpp - a self-reconfigurable data processing architecture." *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[5] Silicon Hive, Philips, http://www.silicon-hive.com.

[6] N. Bansal, S. Gupta, N. D. Dutt, A. Nicolau, and R. K. Gupta, "Interconnect-aware mapping of applications to coarse-grain reconfigurable architectures." in *FPL*, 2004, pp. 891–899.

[7] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays," in *ASAP'05*, 2005, pp. 161–168.

[8] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel, "A new design approach for processor-like reconfigurable hardware," in *Euro DesignCon*, Munich, Germany, 2004.

[9] T. Oppold, T. Schweizer, T. Kuhn, W. Rosenstiel, U. Kanus, and W. Straßer, "Evaluation of ray casting on processor-like reconfigurable architectures." in *FPL*, 2005, pp. 185–190.

[10] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel, "A design environment for processor-like reconfigurable hardware," in *IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, Dresden, Germany, 2004, pp. 171–176.

[11] J. A. Brenner, "Simultaneous scheduling, binding and routing for processor-like reconfigurable architectures," Master's thesis, Braunschweig University of Technology, 2005.

[12] S. Chaudhuri, R. A. Walker, and J. E. Mitchell, "Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem," *IEEE Transact. VLSI Systems*, vol. 2, no. 4, pp. 456–471, 1994.