A graph-coloring approach to the allocation and tasks scheduling for reconfigurable architectures

Marco Giorgetta Marco Santambrogio Donatella Sciuto Paola Spoletini

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5
20133 Milano, Italy

{giorgett,santambr,sciuto,spoleti}@elet.polimi.it

ABSTRACT

Designing systems mapped onto FPGAs that foresee a dynamic reconfiguration of the application is a difficult task. It requires that the identification of the reconfigurable tasks and their allocation onto the FPGA must be defined during the design phases. Furthermore, also the schedule of dynamic reconfigurations must be defined. This paper presents an improved scheduling and allocation of reconfigurable tasks onto an FPGA, based on the coloring problem. The proposed algorthm stems from the one previously presented [1], but introduces backtracking to improve the performance in terms of number of number of colors, that represent FPGAs areas. The new algorithm has been experimented on the Xilinx-based architecture defined to support dynamic reconfigurability [2].

1. INTRODUCTION

Most applications running on FPGA-based systems are implemented using a single configuration per FPGA, an example of this solution is SPLASH, [3]. This means that the functionality of the circuit does not change while the application is running. Such an application can be referred to as being Compile-Time Reconfigurable, CTR, because the entire configuration is determined at the compile-time and does not change throughout system operation. Another implementation strategy is to implement an application with multiple configurations per FPGA [4], [5], [6]. In this scenario the application is divided into time-exclusive operations that need not, or cannot, operate concurrently. Each operation is implemented as a distinct configuration which can be donwloaded into the FPGA as necessary at run-time during application operation. This approach is referred to as Run-Time Reconfiguration, RTR or Dynamic Reconfiguration.

In [7] the authors propose a new methodology to allow the platforms to hot-swap application specific modules without disturbing the operation of the rest of the system. This goal is achieved through the use of partial dynamic reconfiguration. The application presented in that paper has been implemented onto a Xilinx Virtex-E FPGA. According to this, the proposed methodology finds its physical implementation as an external reconfiguration that implies that a Virtex-E active array may be partially reconfigured by an external device such as a Personal Computer, while ensuring the correct operation of those active circuits that are not being changed [8]. The reconfigurable modules are called Dynamic Hardware Plugin, DHP. The methodology proposed in [7] transforms standard bitfiles, computed by common computer aided design tools, into new partial bitstreams that represent the DHP modules due to the PARtial BIt-file Transform tool, PARBIT, [9]. The PARBIT tool transforms FPGA configuration bitstreams to enable Dynamically Hardware Plugins modules in the Field-programmable Port Extender, FPX, [10]. The tool accepts as input the original bitfile, a target bitfile and parameters given by the user and provides as output the new bitstream which can load a DHP module into any region of the Reprogrammable Application Device, RAD on the FPX.

In [11], the authors considered the reconfigurable computing as a close combination of hardware cores and of the run-time instruction set of a general purpose processor. The classification of core types is generally accepted to be split into three classes [12]: Hard cores, Firm cores and Soft cores.

In [13], a new class of cores called run-time parameterizable (RTP) has been introduced. RTP cores allow a single core to be computed and customized at run-time. For example, an adder core can be produced, and then parameterized at run-time for different operand widths. The core produces all the required configuration data to define the logic and the routing. The possibility of determining limited amounts of routing at run-time is also dealt with in [13]. An innovation of this approach consists in considering the RTP cores as a specific example of a reconfigurable core, placed on the programmable device in a dynamic manner to respond to the changing computational demands of the application. The problem of this methodology is that the RTP are targeted only to a single device family and there is no information about the communication channel between RTP and about how they solve the physical reconfiguration problem. To control the mapping of cores at application run-time onto the programmable device, a management mechanism is required. The algorithm proposed in this paper is part of the Caronte methodology, [2, 14] for the dynamic reconfiguration of an embedded system introducing a partial dynamic reconfiguration degree in the design phase. Starting from a previous solution proposed in [1] this work extends that solution by providing a more flexible algorithm, specifically tailored for the Caronte architecture, [2], which is able to find better solutions, considering the number of colors found, for the graph coloring problem, meeting the timing constraints of the considered architecture, where the placement of each module has to be statically defined.

2. THE CARONTE METHODOLOGY

As proposed in [2, 14], the Caronte Flow is mainly composed of three phases:

HW-SSP Phase The HardWare Static System Photo Phase is based on the EDK, [15], tool. It identifies a set of EDK system descriptions, the static descriptions, that will be used to define all the necessary reconfigurations;

Design Phase This phase aims at creating all the information needed to compute all the bitstream to physically implement the embedded reconfiguration of the FPGA. This phase solves three different problems:

- Identify the structure of each reconfigurable block providing a specific implementation for each of them. This phase is based on the Xilinx Modular Based Design approach;
- Identify, using the *Floorplanner* tool provided in the ISE tool chain, the area of each reconfigurable component of the system;
- Solve the communication problem between reconfigurable modules, by introducing *Bus Macros* that allow signals to cross over a partial reconfiguration boundary.

Bitstream Creation Phase This phase creates all the bitstreams needed to implement the system description onto an FPGA through the dynamic embedded reconfiguration.

The reasons why a designer could be interested in a dynamic system implementation could be different, but the implementation problem remains the same: the system description must be partitioned in a fixed set of components that have to be dynamically mapped onto an architecture, that has been partitioned too. Both the FPGA and the initial description of the system have to be partitioned into several parts to provide the correct starting point to find out a dynamic reconfigurable design for the desired system description. This first phase, solved by the proposed algorithm, identifies all the processing elements of the description that will be mapped onto the corresponding part of the FPGA, see Figure 1.

These elements, in order to be downloaded onto an FPGA, have to be transformed into a set of reconfiguration bitstreams by the Caronte Flow. Figure 1 shows the *logical partitioning layer* used in the proposed methodology where:

Task Graph Layer Is the input of the Caronte Flow provided by the System Partitioning Phase;

EDK Layer All the processing elements computed in the first phase have to be translated into a reconfigurable system component description that can be used to compute the bitstream that will be downloaded onto the FPGA;

FPGA Layer The FPGA has to be divided into parts that will become the sites for the reconfiguration elements computed by the previous phases.

3. THE GRAPH COLORING SYSTEM

A legal vertex coloring of a graph $G = (V, E)$ is an assignment of colors to its vertices such that no two adjacent

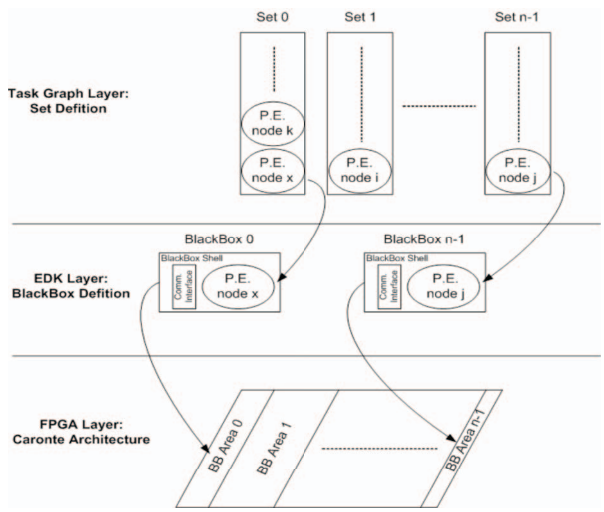

Figure 1: Partitioning layers

vertices share the same color. Equivalently, a legal coloring of G by k colors is a partition of its vertices into k disjoint sets. Formally, the graph coloring problem is defined as in the following.

Let $G = (V, E)$ be a graph. A *coloring* of G is a map $\theta : V \to N$ of nodes to colors such that any two adjacent vertices have different colors.

In this paper the graph coloring problem is solved for a conflict graph associated with a given scheduling of an algorithm, where each node represents a task to be executed and each edge represents a scheduling conflict — that is, two tasks cannot be executed at the same time on the same piece of FPGA. The colors will then represent different areas of the FPGA. In the context of the proposed architecture, however, two novelties arise.

Reconfiguration First of all, reconfiguration complicates the constraints. Considering the same example proposed in [1], Figure 2 (a) shows a colored conflict graph (the labeling is chosen in such a way that for every m, n the node m/n is data–dependent on $m/(n-1)$).

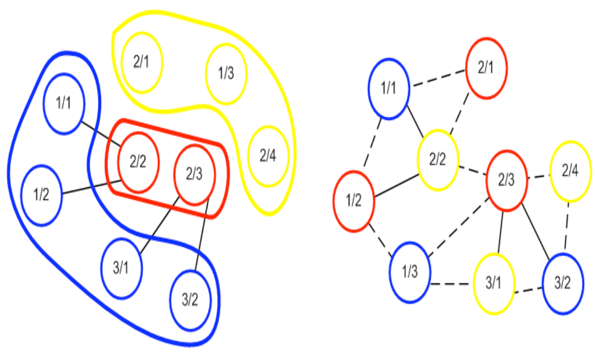

Figure 2: (a) Coloring of a conflict graph, [1]; (b) New Coloring Solutions

Consider, though, what happens when mapping this solution on the FPGA: as shown in Figure 2 (a) both 1/1 and 1/2 have been assigned to the same FPGA area. This means that at the end of the execution of block 1/1, the system has to wait till block 1/2 is mapped on this area to proceed, and the loading can't be done in parallel with the execution of 1/1 since they both need the same area. This problem is indeed serious, since reconfiguration times are still significant with respect to execution times (for instance a 548 kB bitstream file needs 28 seconds for the external reconfiguration). Hence to properly solve the scheduling problem there is another information that needs to be taken into account: the *reconfiguration* conflicts.

In order to do so a conflict graph variant is introduced, adding data dependency edges to the original conflict graph. This new graph is similar to an over–constrained conflict graph that forces a different coloration for two nodes having a data dependency, and is called *Task Conflict Graph*, TCG.

The TCG can now be colored to find the new bindings; an admissible coloring is shown in Figure 2 (b).

Online processing Secondly, the coloring task is performed in two very different situations, as explained in Section 2: not only *statically* at compile–time, but also *dynamically* at run–time if some BlackBox execution time is larger than expected. This poses an additional constraint: if at design–time the coloring algorithm is allowed to have long runtimes in order to determine a better solution (fewer colors), at run–time this is not possible, since otherwise the execution of the tasks waiting to be started will be delayed unacceptably.

What is needed in this architecture, then, is a very fast (and of course reasonably accurate) graph coloring algorithm.

3.1 The Adj algorithm, previous version

`Adj` [1], was designed to solve the coloring problem very rapidly while still retaining a good quality of the solution.

The approach of the `Adj` algorithm is to scan all the nodes, coloring at each iteration not only the node v being considered, but also all its neighbouring nodes $\texttt{Nb}(v) := \{w \in V | \exists (v,w) \in E\}$, unless they are already colored. If they are, it checks for color conflicts. If during the scan of the neighbors of node v the algorithm finds that $w \in \texttt{Nb}(v)$ is colored and has a color conflict, this conflict is signalled setting w's color to -1, and it will be dealt with in the iteration considering node w.

The `Adj` algorithm pseudo code is presented in Algorithm 3.1. The choice of the actual color being assigned to a node is done via an heuristic based on the $\texttt{Friend}(c, Cols)$ function, which among the colors $Cols$ chooses the one that is most frequently adjacent to c in the graph at a particular time.

The `Friend` function checks how many edges exist with certain endpoint colors. Its complexity is therefore $O(|E|)$. The `CheckConflict` function, instead, has $O(|V|)$ worst–case complexity (for a dense graph). As for the `ColorThis` function, steps 1. and 2. require $O(|V|)$ while the else branch takes $O(|V|)$ plus the complexity of `Friend`. Hence we obtain $O(|V| + |E|)$. It is then easy to see that `ColorNb` takes $O(|V|(|V| + |E|))$. From this it can be deduced that an upper bound for the worst–complexity of the overall algorithm is $O(|V|^2(|V| + |E|))$. This bound is not tight at all, though, since it assumes that condition (a) in the main body is always true. Obviously this is not the case, since some coloring is performed by `ColorNb` executed for previous nodes. In the case of sparse graphs (as those arising from our problem always are), however, assuming that the cardinality of every neighborhood is bound by a constant, one gets $O(1)$ for `CheckConflict`, $O(|E|)$ for `ColorThis` and hence for `ColorNb`, so that the total complexity is $O(|E||V|)$.

Algorithm 3.1: The `Adj` algorithm pseudocode.

1 $AllColors \leftarrow \emptyset$;
2 $Colors \leftarrow (0, \ldots, 0) \in \mathbb{N}^{|V|}$;
3 **foreach** $v \in V$ **do**
4   **if** $Colors[v] == 0$ **then**
5   `ColorThis` (v);
6   `ColorNb` (v);
7   **endif**
8   **if** $Colors[v] == -1$ **then**
9   ColorThis(v);
10   **endif**
11   ColorNb(v);
12 **endfch**

13 function `ColorThis` (node v)
14 $ExcludedColors \leftarrow \bigcup \{Colors[w] | w \in Nb(v)\}$;
15 $AvailColors \leftarrow Colors \setminus ExcludedColors$;
16 **if** $|AvaiColors| == 0$ **then**
17   $AllColors \leftarrow AllColors \bigcup \{|AllColors| + 1\}$;
18   $Colors[i] \leftarrow |AllColors|$;
19 **endif**
20 $c \leftarrow \text{argmax}_c \, |w \in Nb(v) | Colors[w] = c|$;
21 $Colors[v] \leftarrow \texttt{Friend}(c, AvailColors)$;

22 function `ColorNb` (node v)
23 $c \leftarrow \texttt{Friend}(Colors[v], AllColors \setminus$
24     $\{Colors[w] | w \in Nb(v)\})$;
25 **foreach** $w \in Nb(v)$ **do**
26   **if** $Colors[w] \neq 0$ **then**
27   `ColorThis` (w);
28   **endif**
29   **if** *`CheckConflict` (w)* **then**
30     $Colors[w] \leftarrow -1$;
31   **endif**
32   **foreach** $w \in Nb(v)$ **do**
33     $Colors[w] \leftarrow c$;
34   **endfch**
35 **endfch**

36 function `CheckConflict` (node v)
$ForbiddenColors \leftarrow \bigcup \{Colors[w] | w \in Nb(v)\}$;
37 return $(ForbiddenColors \bigcap \{Colors[v]\} == \emptyset)$

38 function Friend(color c, colors Cols)
39 **if** $Cols == \emptyset$ **then**
40   $AllColors \leftarrow AllColors \bigcup \{|AllColors| + 1\}$;
41   return $|AllColors|$;
42 **endif**
43 return $\text{argmax}_{f \in Cols} \, |\{w | w \in Nb(v)$
44     $\wedge Colors[v] = c \wedge Colors[w] + f\}|$;

3.2 Algorithm Modification: Backtracking

When the `Adj` assigns a color to a node, it might have more than one choice of color that can be assigned to the node without generating a conflict. If there is no possibility to use an already introduced color, i.e. this set of choices is empty, then a new one must be inserted in the coloring. Namely, a new color can be inserted when coloring the adjacencies of the first node, which is obviously unavoidable, and when coloring a node whose adjacencies have already taken all of the colors in the pool of used colors.

When adding a new color in the second case, chances are that if in a previous color assignment another available color had been chosen, the current node could be assigned to an already used color instead of a new one. This suggests the idea of applying a backtracking technique to the `Adj` algorithm.

As we are dealing with *directed acyclic graphs*, there always exists a partial order on the vertices. Performance - and quality of the results - may vary significantly depending on the particular total order in which the vertices are processed. Hence, we propose to process the nodes from the ones belonging to the denser subgraphs moving on to the ones belonging to the sparser subgraphs, always respecting the partial order.

After having outlined the main idea, and defined the order in which vertices will be colored, let us define a few expressions that will be used in the following.

The **main computation** is the first instance of the algorithm, launched on the completely uncolored graph, that may launch a backtracking procedure in order to reduce the number of used colors.

A **legal color** for node i is a color that can be assigned to node i without generating a conflict.

The **freedom set** of node i, is the set of legal colors that can be assigned to node i. Note that the colors in this set are the legal colors at the time that node i was colored, since when the algorithm backtracks to a previously taken choice, it must be able to set node i to another color among those that were allowed at its previous coloring.

The **BackSet** of level n of node i, is composed of all the n-hop distant nodes with non empty freedom set.

The **state** of the coloring is the union of the colors of the nodes and the freedom set of each node.

The possibility to backtrack is evaluated at the steps the `Adj` inserts a new color, when it is trying to solve the conflict of current node, or color the current node.

In such cases, the behavior of the `AdjB` is the following. Let i be the node that is causing the introduction of a new color. The BackSet for node i is built, according to the level (which is user-defined). Then the BackSet is ordered since the backtracking will be performed from the closest node to the farthest. Subsequently, we try to backtrack to a node in the BackSet (from the closet to the farthest) until a better solution is found or the BackSet has been completely used (in such case, backtracking does not produce any better solution). Backtracking to node $j \in BackSet(i)$ means setting that node to another color among the ones in j's freedom, and starting the coloring from node j up to node i. Once node i has been reached, if the number of used colors is less than the used colors in the main computation then the state of the computation is replaced by the result state of the backtrack operation. During the computation, in case the number of colors increases over the number of used colors in the main computation, the backtrack operation is simply interrupted.

If all the colors in the freedom set of all the nodes in the BackSet have been tried and no improvement has been achieved, then the backtrack did not lead to a better solution, and the color is added to the pool of used colors. Otherwise, a color could be saved; in any case the coloring procedure goes on.

The pseudocode of the modified algorithm is presented in Algorithm 3.2. Note that throughout the algorithm, all the commands (such as assignment of a color to a vertex) are implicitly executed on the *current state* of the graph, which includes, as explained before, the state of each node as well as the freedom sets and back sets of the nodes.

Accordingly to the notation used in the above explanation, the node that is causing the introduction of a new color is node i, while the node we are backtracking to is node j.

Algorithm 3.2: The `AdjB` algorithm pseudocode.

```
1  Main: called upon insertion of a new color in Adj
2  verticesList ← orderedVertices(BackSet(i));
3  foreach node v ∈ BackSet(i) do
4      foreach node u ∈ FreedomSet(v) do
5          backtrackState ← buldColoringState (v);
6          backtrackSstate ← doColor
           (backtrackState);
7          if |UsedColors(backtrackState)| <
           |UsedColors(currentState)| then
8              currentState ← backtrackState;
9              break;
10         endif
11     endfch
12 endfch

13 function buldColoringState (node i)
14 foreach color c ∈ FreedomSet(i) do
15     if c ≠ Color[i] then
16         Color[i] ← c;
17         FreedomSet(i) ← FreedomSet(i) \ {c};
18         break;
19     endif
20 endfch

21 function doColor (graph_state state)
22 foreach node w from j to i do
23     if Color[w] ≠ 0 then
24         foreach u ∈ Adjacency(w) do
25             if u < j then
26                 continue;
27             endif
28             if Color[u] ≠ 0 then
29                 if Color[u] = Color[w] then
30                     Color[u] = -1;
31                 endif
32             endif
33             Color[u] ← ColorThis (u);
34         endfch
35     endif
36     Color[w] ← ColorThis (w);
37 endfch

38 function ColorThis (node w)
39 –see Adj for ColorThis function
```

The *main* pseudo code portion, from line 1 to line 12, is

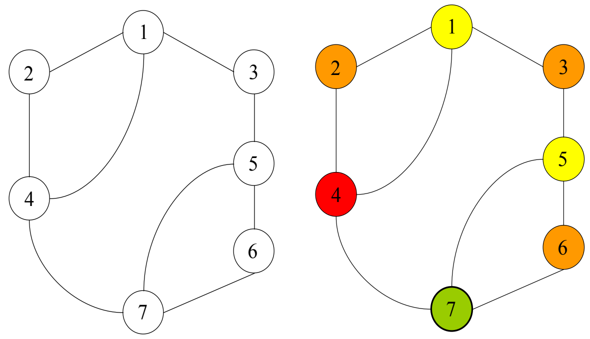

Figure 3: Graph used as example, with the coloring obtained applying the `Adj` algorithm.

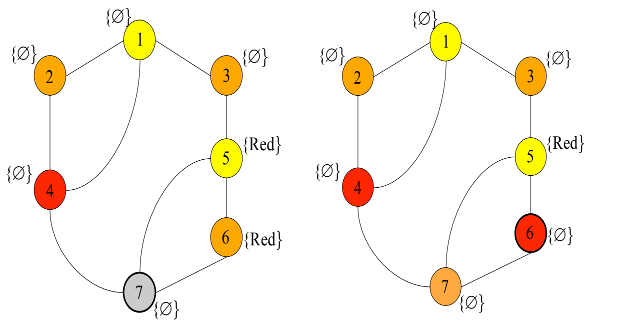

Figure 4: The graph of Figure 3, with the FreedomSet of each node, colored with the backtracking technique.

run when the `Adj` algorithm tries to insert a new color in the set of used colors. The algorithm goes backwards in the total order of colored nodes so far; it builds a backtracking temporary state of the graph by assigning to a node in the backset a legal color in its freedom set, and resumes the coloring from that node.

Notice that the `doColor` function performs a coloring of the node and adjacencies just like the `Adj`'s ColorThis does; but, unlikely the latter, it does not affect the nodes that are lower than j in the considered total order. This happens because in case we mark a conflict in a node lower than j, then we are not going to solve that conflict, since the node will not be processed by the main computation anymore (unless, of course, another backtracking run is performed on a node even lower than the marked one, but this is unpredictable).

It must be noticed that when the `ColorThis` function is called in a "backtracking step", and it introduces a new color to the set of used colors, the backtracking is not recursively applied.

Figure 3 shows an example of a simple graph, whose coloring is improved by the adoption of our backtracking solution. Indeed, the `Adj` algorithm uses three colors for this graph, since it solves the conflict that arises on node 7 by introducing a new color. Instead, as shown in Figure 4, the backtracking version of the `Adj` algorithm, before introducing the new color, tries to assign to a previously colored node with non empty freedom set another legal color. In Figure 4, next to each node its freedom set is indicated. Hence, when the `Adj` algorithm would add a new color in order to solve the conflict on node 7, the `AdjB` algorithm backtracks to node 6, whose freedom set is non empty, and chooses the other coloring option, i.e. red. This allows to color node 7 in orange, thus saving the introduction of a new color.

4. RESULTS AND ANALYSIS

The `AdjB` algorithm has been implemented in C, and timed on an Intel PIII running at 1GHz. In the current implementation, in all the attempts of introduction of a new color, the backtracking technique is currently called. While this leads to an improvement in the number of colors adopted for the graph coloring, it also implies a sensible loss in terms of performance, due to the fact that the backtracking does not always provide gains. This issue will be solved with the introduction of a *backtracking confidence function*, that shall evaluate the probability of success of a backtracking call keeping into consideration the previous calls and the density of the sub graph surrounding the node that is requiring a new color: the denser the graph is, the less likely we will be able to get an improvement. Note that the implementation of this confidence function has been left to future developments.

Anyway, preliminary results indicate a sensible improvement in the number of used colors against the number computed by the `Adj` algorithm. As in [1], tests have been conducted on the instances of the graph coloring problem for DIMACS computational challenge. The results for some of the graphs are shown in Figure 6. As it can be noticed, there is no explicit correlation between the improvement the number of used colors and the size of the graph, since the variance of the color gain is below 1% of the average (0.0034 against 0.87).

The maximum level of backtracking for these tests has been set to two, which means that in the backtracking color choice affects only the 2-hop-distant nodes. Such a conservative setting of the algorithm has been made due to the lack of the backtracking confidence function; once this feature is implemented, the `AdjB` algorithm can be set to affect a broader sub-graph when attempting to avoid a new color.

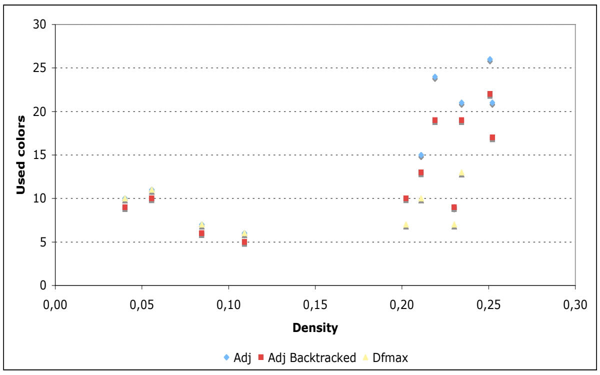

Figure 5: Colors required for different densities

Figure 5 plots the number of used colors against the graph density for some graphs colored by the Dfmax algorithm, the

Graph	Nodes	Edges	Density	Adj	Adj Time	Dfmax	AdjB	AdjB Time	Color Gain
jean	80	254	0,04	10	0.1	10	9	0.18	0,90
huck	74	301	0,06	11	0.11	11	10	0.2	0,91
myciel6	95	755	0,08	7	0.96	7	6	2.4	0,86
myciel5	47	236	0,11	6	1	6	5	0.24	0,83
queen7_7	49	476	0,20	12	0.1	7	10	1.46	0,83
queen9_9	81	1368	0,21	15	0.17	10	13	4.45	0,87
queen14_14	196	8372	0,22	24	1.13		19	55.3	0,79
queen6_6	36	290	0,23	9	0.1	7	9	0.38	1,00
queen13_13	169	6656	0,23	21	0.78	13	19	39.41	0,90
queen15_15	225	12640	0,25	26	1.51		22	98.2	0,85
queen12_12	144	5192	0,25	21	0.51		17	35.6	0,81

Figure 6: Some test results.

Adj and the AdjB algorithm. As expected, AdjB results find place between the Dfmax and the Adj ones, as it performs corrective interventions on the Adj coloring.

5. CONCLUSIONS

The paper has introduced a new algorithm for the coloring problem tackled to solve the reconfiguration problem in the Caronte design methodology. Adj the previously introduced algorithm, used the information both on the node to be colored and on the neighbouring nodes. This solution is very fast but can be improved in terms of number of colors introduced, improving the overall scheduling procedure. Obviously an exact method to obtain the optimal number of colors is not acceptable in terms of computation time. The algorithm, proposed in this paper, the AdjB , is a good compromise between these two, in fact, as it is possible to notice from the experimental results, it reduces the number of colors, found by Adj by preserving an acceptable the computation time. AdjB exploits the idea of Adj but, before introducing a new color, reconsiders the assigned colors with a backtracking procedure to check if this addition is unavoidable. This modification in the algorithm reduces the number of colors needed to color the graph, favoring the performance of the overall execution of the considered program, meeting the timing constraints of the considered architecture.

6. REFERENCES

[1] Fabrizio Ferrandi, Massimo Redaelli, Marco D. Santambrogio, and Donatella Sciuto. Solving the coloring problem to schedule on partially dynamically reconfigurable hardware. In *IFIP VLSI-SOC 2005*, 2005.

[2] Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, Marco D. Santambrogio, and Donatella Sciuto. Caronte: a complete methodology to implement partially dynamically self-reconfiguring embedded systems on modern fpga. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, 2005.

[3] D. T. Hoang. Searching genetic databases on splash2. pages 185–191. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, 1993.

[4] D. Ross, O. Vellacott, and M. Turner. An fpga-based hardware accelerator for image processing. pages 299–306. More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications, W. Moore and W. Luk, 1993.

[5] P. Lysaught, J. Stockwood, J. Law, and D. Girma. *Artificial neural network implementation on a fine-grainde FPGA*. R. Hartenstein and M.Z. Servit, 1994.

[6] P.C. French and R.W.Taylor. A self-reconfiguring processor. pages 50–59. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machine, D.A. Buell and K.L. Pocek, 1993.

[7] Edson L. Horta, John W. Lockwood, and David Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfigurtion. pages 844–848, 1993.

[8] S. Tapp. Configuration quick start guidelines. *XAPP151*, July 2003.

[9] Edson Horta and John W. Lockwood. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). *Washington University, Department of Computer Science, Technical Report WUCS-01-13*, July 2001.

[10] David E. Taylor, John W Lockwood, and Sarang Dharmapurikar. Generalized rad module interface specification of the field programmable port extender (fpx). *Washington University, Department of Computer Science. Version 2.0, Technical Report*, January 2000.

[11] G. Brebner. A virtual hardware operating system for the xilinix xc6200. pages 327–336. IEEE Symposium on Field Programmable Logic and Applications, 1996.

[12] J. Case, N. Gupta, L.J. Mitta, and D. Ridgeway. *Design methodologies for core-based FPGA design.* Xilinx Inc., 1997.

[13] S. Guccione and D.Levi. Run-time parameterizable cores. pages 215–222. IEEE Symposium on Filed Programmable Logic and Application, 1999.

[14] Alberto Donato, Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. Operating system support for dynamically reconfigurable soc architectures. In *IEEE-SOCC*, 2005.

[15] Xilinx Inc. *Embedded Development Kit EDK 6.2i.* Xilinx Inc., 2004.