Maximum Edge Matching for Reconfigurable Computing

Markus Rullmann and Renate Merker
Dresden University of Technology, Germany
Circuits and Systems Laboratory
{rullmann, merker}@iee1.et.tu-dresden.de

Abstract

Reconfiguration of tasks implies considerable overhead on the amount of configuration data and time. Much overhead is caused by redundant configuration generated by the design tools which implement similar structures in the designs on different resources. In this paper we propose a new method to identify structural similarities in tasks. Based on this information, we are able to generate automatically constraints to ensure that the place and route tools use identical resources. Thus we ensure that less redundant configuration is produced. In this paper we give a formal description of the underlaying maximum edge matching problem *and show a method to solve it optimally. We derive a truncation criteria to restrict the search space efficiently. We also propose an Ant Colony Optimization based solution with a problem specific local heuristic and show that it performs optimal as well in our examples, but with considerable lower computational effort.*

1 Introduction

Reconfigurable computing has received an ever increasing interest in the research community in recent years. The associated problems cover all aspects of digital system design, but a major driver is the potential in flexibility, performance and power consumption of such systems. A disadvantage is still the amount of device programming data required to reconfigure devices at runtime. The configuration data needs to be stored in the system memory and transferred to the programming interface of the reconfigurable devices. As such a large amount of configuration data has a negative impact on system cost, reconfiguration time and power dissipation.

Several methods to reduce reconfiguration costs have been proposed. Generally, different tasks have to be implemented in a device (e.g. FPGA). Due to a lack of area the corresponding modules on a device have to be reconfigured. Many authors consider task scheduling and prefetching techniques to hide reconfiguration time from the application [4, 5, 2, 8, 6]. However, reduced configuration data and power can be achieved by the use of custom reconfigurable architectures [3, 14] or by advanced implementation methods that exploit similar properties of tasks. In [9] a method that uses remapping of LUT based logic in FPGAs is proposed. This method does not optimize routing configuration which is dominant in todays FPGAs. Rakhmatov et al. [10] proposes an algorithm to identify common routing channels in a design. The approach requires a design partitioning into functional and bus logic and can not utilize common routes inside the datapath units. Shirazi et al. [12] perform a matching of components by using heuristic weights for routing and placement similarities. The method produces maximum component matches but a maximum routing match can not be guaranteed. In [1] structural similarities are identified by common subgraph extraction, which exploits only similarities represented by the largest common subgraph. In our approach we search for structural similarity of the different tasks, i.e. for such parts that can be both mapped to the same resource types and placed and routed using the same physical resources. Moreane et al. [7] uses a similar approach for the design of reconfigurable datapath architectures.

Our approach targets at the mapping of tasks at netlist level to a predefined reconfigurable architecture. Generally tasks are implemented by the placement and routing tools independent of each other. Hence the tools generate different configurations for similarly structured logic in each of the tasks. When such tasks are reconfigured, a large amount of configuration data is used to implement the same similarly structured logic on different physical resources. We call this *redundant configuration.* The overhead associated to reconfigurable systems can be reduced by eliminating the redundant configuration.

We consider fine grain reconfigurable systems. From a lot of experiments follows that in these systems most configuration data is spend on the routing configuration. This is partly due to the fact that there exists a number of possible routes between two physical resources. The target of the routing tool is just to find one possible route that meets the design constraints.

To reduce redundant configuration we search for structural similarity between two tasks first. Secondly, this information guides the place and route tools to occupy the same physical resources in the modules. In [11] we described how modules can be especially designed to improve structural similarities.

This paper proposes a novel method to extract structural similarities in two given tasks at netlist level. The structural similarities describe exactly which connections between instances in one task are equivalent to connections in the other task. The information is used to map these instances of both tasks onto the same physical resources and to implement the routing of the equivalent connections on the same physical resources, too. Therefore similar structures in both tasks do not need reconfiguration because the constraints during place and route ensure that they will have an identical configuration. The method can be integrated into existing FPGA design flows directly by the use of our automated optimization tool.

The rest of the paper is structured as follows: The netlists are transformed to an equivalent graph model (Section 2) that can be treated more formally using the methods described in Section 3. The computational complexity of the underlaying problem is too high to be solved optimal. We developed an Ant Colony Optimization method (Section 4) that has optimal performance in our examples in Section 5.

2 Graphs for Modeling Netlists

A netlist consists of instances and nets connecting the ports of these instances (see Figure 1(a)). Instances can be modelled as nodes in a graph. Each net is represented by a number of directed edges in the graph: Each directed connection in the net between the output port of a driving and input ports of receiving instances is represented by a directed edge in the graph. In addition, all nodes representing instances that are mapped to the same physical resource type belong to the same *class of nodes*. In Figure 1 an example for a netlist (a) and the equivalent graph (b) is shown. The instances are mapped to two types of physical resources (white and gray shaded), dividing the nodes into two classes: $\{0, 1, 3\}$ and $\{2, 4\}$. Net 1 connects the output port of instance 1 with the input ports of the instances 1,2,3 and 4, while net 2 connects the output port of instance 0 with the input port of instance 1, hence the graph contains 4 edges starting at node 1 and one edge from node 0 to node 1.

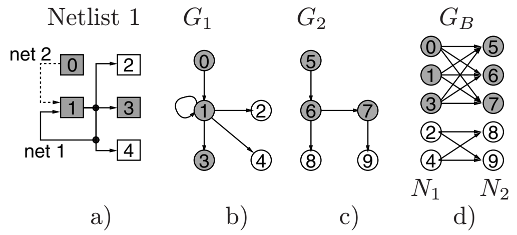


Figure 1. Two example input graphs G_1, G_2, the netlist 1 for G_1 and the resulting bipartite graph G_B.

3 Maximum Edge Matching Problem

In this section we introduce the maximum edge matching problem. We derive a specialized branch and bound method to find a globally optimal solution for this problem.

3.1 Problem Definition

At first we define $G(N, E)$ as a finite digraph with a set of nodes N and a set of edges E. A directed edge from node $r \in N$ to node $s \in N$ is denoted as (r, s). Two digraphs $G_1(N_1, E_1), G_2(N_2, E_2)$ are considered as an input to the edge matching problem.

We define a complete bipartite graph $G_B(N_1 \cup N_2, E_B)$ with $\forall (r, s) \in E_B.\ r \in N_1 \wedge s \in N_2$. We assume that $N_1 \cup N_2 = \bigcup_{n=1}^{m} S_n$, where S_n are disjoint sets. All edges $(r, s) \in E_B$ must satisfy the condition:

$$r, s \in S_n \wedge r \in N_1 \wedge s \in N_2. \qquad (1)$$

Hence follows, the bipartite graph $G_B(N_1 \cup N_2, E_B)$ consists of m bipartite subgraphs $G_n(S_n, E_S)$ where $(r, s) \in E_S$ if condition (1) holds.

In Figure 1(b,c) two input graphs with five nodes each are shown. The nodes are classified into the sets S_1 and S_2 (gray and white filled respectively). The resulting bipartite graph G_B has two bipartite subgraphs.

Within the bipartite subgraphs, a graph matching can be performed. A matching is a set $M \subseteq E_B$ of non-adjacent edges in G_B. Two edges $(r, s) \in E_B$ and $(r', s') \in E_B$ with $(r, s) \neq (r', s')$ are adjacent, if $r = r'$ or $s = s'$.

Definition 1 *The edge $(r,s) \in E_1$ is a matching edge if there is an edge $(r',s') \in E_2$ such that $(r,r') \in M$ and $(s,s') \in M$. $M_{E_1^M} \subset E_1$ is the set of matching edges of E_1.*

Considering the example in Figure 1 and a given matching $M = \{(0,5),(1,6),(3,7),(2,8)\}$, we get $M_{E_1^M} = \{(0,1),(1,3),(1,2)\}$.

Definition 2 *The matching weight of matching M is $w_M = \mid M_{E_1^M} \mid$.*

The maximum edge matching problem consists in finding a matching $M_{Max} \subseteq E_B$ that results in a maximum matching weight.

The maximum matching M_{Max} describes the structural similarity of the two input netlists. The matching weight of M_{Max} defines the number of routes between the instances of the netlist that should use the same routing configuration. The instances represented by the nodes $(r,s) \in M_{Max}$ must be placed on the same physical resources, too.

3.2 Complexity of the Maximum Edge Matching Problem

Obviously M_{Max} is a subset of one of the maximum matchings M_P in G_B. A matching M is a maximum matching M_P, if M_P contains the maximum possible number of elements where

$$\mid M_P \mid = \sum_{n=1}^{m} \min(\mid N_1 \cap S_n \mid, \mid N_2 \cap S_n \mid).$$

Hence follows for finding M_{Max} all maximum matchings M_P have to be investigated. The set of all M_P in G_B is denoted as A_P and the set of all maximum matchings M_{S_n} in G_n as A_{S_n}. The total number of maximum matchings M_P can be determined as follows.

The number of possible maximum matchings M_{S_n} in a bipartite subgraph G_n is:

$$\begin{aligned} \mid A_{S_n} \mid \quad &= \quad (i)_j = i(i-1)\dots(i-j+1) \\ \text{with} \quad & i = \max(\mid N_1 \cap S_n \mid, \mid N_2 \cap S_n \mid) \\ & j = \min(\mid N_1 \cap S_n \mid, \mid N_2 \cap S_n \mid). \end{aligned}$$

The total number of maximum matchings in G_B is then:

$$\mid A_P \mid = \prod_{n=1}^{m} \mid A_{S_n} \mid . \qquad (2)$$

In the given example (Figure 1) $\mid A_P \mid$ is:

$$\mid A_P \mid = (3)_3 \cdot (2)_2 = 6 \cdot 2 = 12.$$

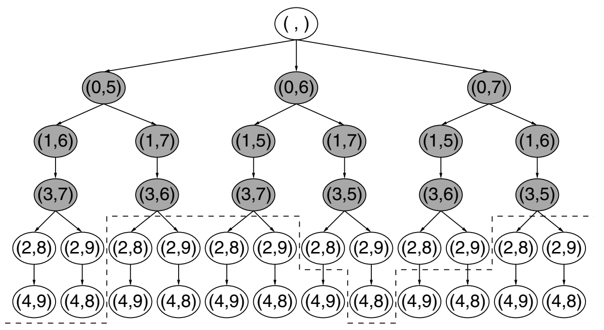


Figure 2. The search tree T for the example from Figure 1. The vertices below the dashed line can be excluded from the search by the truncation criteria.

3.3 An Algorithm for Finding M_{Max}

It is now apparent that a search for M_{Max} can simply enumerate all M_P matchings. However, this will be impossible for most practical problems. We derive an efficient measure to establish a truncated search. It ensures that all M_P matchings are excluded from the search if they can not provide a better solution than the currently best one found.

The enumeration of the maximum matchings M_P can be implemented as a depth–first search on the tree T containing all possible matchings $M \subseteq E_B$. Each path from the root vertex to one of the leaf vertexes represents a possible maximum matching M_P.

In Figure 2 an example search tree $T(N_T, E_T)$ with vertices N_T labeled $(r,s) \in E_B$ is shown. We already calculated the number $\mid A_P \mid = 12$ in Section 3.2, which is equal to the number of leaf vertices in T.

Suppose that at any stage in the depth–first search a matching M is selected. M contains all edges E_B selected by the path from the root vertex to a vertex $(r,s) \in N_T$. If it can be proven that $M \not\subset M_{Max}$ than none of the M_P with $M \subset M_P$ can be M_{Max}. Thus the descent in the depth–first search is canceled and all M_P containing the matching M are discarded from the search. This property is used to truncate the search tree for M_{Max} and to reduce the average runtime. The following definitions are needed to derive the truncation condition:

Definition 3 *The set $M_{E_1^C}$ of candidate edges is defined as follows:*

$$M_{E_1^C} = \{(r,s) \in E_1 \mid (r,r') \notin M \vee (s,s') \notin M;\ r',s' \in N_2\}.$$

We can now formulate an upper bound for the matching weight of all maximum matchings $M_P^M \in A_P$ con-

Matching M	w_M	$\mid M_{E_1^C} \mid$	$\mid M_{E_1^M} \mid + \mid M_{E_1^C} \mid$
(0,5)	0	5	5
(0,5)(1,6)	1	3	4
(0,5)(1,6)(3,7)	2	2	4
(0,5)(1,6)(3,7)(2,8)	3	1	4
(0,5)(1,6)(3,7)(2,8)(4,9)	3	0	3
(0,5)(1,6)(3,7)(2,9)	2	1	3
(0,5)(1,6)(3,7)(2,9)(4,8)	3	0	3
(0,5)(1,7)	0	3	3
(0,5)(1,7)(3,6)	0	2	2^S
(0,6)	0	5	5
(0,6)(1,5)	0	3	3
(0,6)(1,5)(3,7)	0	2	2^S
(0,6)(1,7)	1	3	4
(0,6)(1,7)(3,5)	2	1	3
(0,6)(1,7)(3,5)(2,8)	1	1	2^S
(0,6)(1,7)(3,5)(2,9)	2	1	3
(0,6)(1,7)(3,5)(2,9)(4,8)	2	0	2^S
(0,7)	0	5	5
(0,7)(1,5)	0	3	3
(0,7)(1,5)(3,6)	1	2	3
(0,7)(1,5)(3,6)(2,8)	1	1	2^S
(0,7)(1,5)(3,6)(2,9)	1	1	2^S
(0,7)(1,6)	0	3	3
(0,7)(1,6)(3,5)	0	2	2^S

Table 1. Matchings evaluated by the truncated search method. The symbol ^{S} marks the truncation of the search for a subtree.

taining the matching M as a subset:

$$w_{M_P^M} \leq \mid M_{E_1^C} \mid + \mid M_{E_1^M} \mid .$$

The depth–first search can be truncated if condition (3) is true for a matching M constructed while traversing T:

$$\mid M_{E_1^C} \mid + \mid M_{E_1^M} \mid < w_{M_{P,best}}. \qquad (3)$$

$w_{M_{P,best}}$ is the maximum matching weight of all previously evaluated matches M_P.

In Table 1 the truncated search for the search tree in Figure 2 is illustrated. The depth–first search follows the leftmost edge first and computes the upper bound for every new matching M. If condition (3) is met, the descent in the tree is stopped. As it is illustrated in Figure 2 the search tree contains now only 25 vertices. This is a significant reduction compared to the full search tree which has 40 vertices.

Our experiments show a significant reduction in the average runtime using this method. Still, it can not be guaranteed that this method reduces the search problem since it is clearly dependend on the input graphs and the particular order of the search tree.

4 An ACO Algorithm to Solve the Maximum Edge Matching Problem

Even though the truncated search can find the global optimum of larger problem instances than the complete search does, it is still necessary to use a suitable heuristic in most applications. Therefore an heuristic search algorithm based on the Ant Colony Optimization (ACO) metaheuristic has been developed. It has been demonstrated that ACO algorithms are competitive with other heuristic methods on similar problems e.g. the traveling salesman problem (TSP) and quadratic assignment problem (QAP) (see [13] for further references). Our algorithm uses the extensions of the Max–Min Ant System by Stützle et al. [13].

4.1 The ACO Search Space

The ACO is based on a set of agents, so-called ants, which construct different solutions to the maximum edge matching problem. Each ant m has its own limited view of the search space.

At each level l an ant adds one edge $(r, s) \in E_B^l$ with $E_B^0 = E_B$ to the ant's matching. The edge is chosen depending on the associated probability that is calculated from a parameter that resembles the quality of the matching from the previous iterations (pheromone levels τ) and on a parameter based on a local heuristic weight (η). The set E_B^{l+1} for the subsequent level contains all elements from E_B^l that are not adjacent to (r, s) in the graph G_B.

E.g. in level 1 (Figure 3) any $(r, s) \in E_B$ can be chosen. In the example, the selected edge is $(0, 5)$. The level 2 contains all edges from G_B except both $(0, 5)$ and all edges adjacent to $(0, 5)$ in G_B. The graph construction is continued until $E_B^l = \emptyset$. The example shows a path in the search space that leads to a matching similar to the example in Section 3.3.

4.2 ACO Algorithm

The ACO algorithm is an iterative process. In each iteration a number of ants construct different matchings. The quality of the maximum matching is used to update the probability of the edges in the search tree. With an increasing number of iterations, edges that lead to good global solutions strengthen their weights and hence the paths chosen by the ants improve towards an optimal solution. Often this solutions are good local optima of the search problem but a global optimum can not be guaranteed.

The following steps describe an iteration t of the ACO algorithm to solve the maximum edge matching

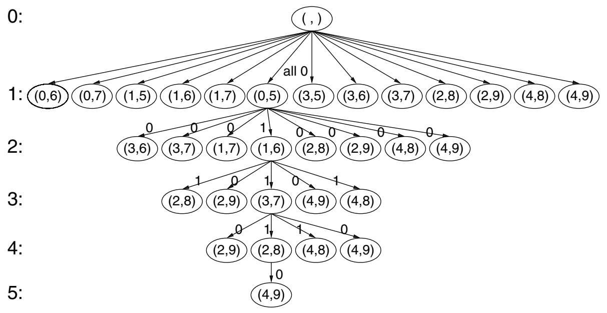


Figure 3. Search tree for constructing a matching with the ACO algorithm.

problem:

1. For each ant k, construct a match $m \in M_P^k$ using the method described in Section 4.1. The probability to choose an edge $(r, s) \in E_B$ in step l from all remaining edges is:

$$p_{r,s}^l(t) = \frac{[\tau_{r,s}(t)]^\alpha \cdot [\eta_{r,s}]^\beta}{\sum_{(r',s') \in E_B} [\tau_{r',s'}(t)]^\alpha \cdot [\eta_{r',s'}]^\beta}$$

   where α, β are tuning parameters for the ACO algorithm. $\tau_{r,s}$ and $\eta_{r,s}$ resemble the pheromone levels and the local heuristic weight for the chosen match, respectively.

2. After all ants have constructed their maximum matching, the pheromone levels are updated based on the rules:

$$\tau_{r,s}(t+1) = (1-\rho) \cdot \tau_{r,s}(t) + \sum_{k=1}^{m} \Delta\tau_{r,s}^k(t) \qquad (4)$$

$$\Delta\tau_{r,s}^k(t) = \begin{cases} w_{M_P^k} & \text{if edge } (r,s) \text{ is used by} \\ & \text{ant } k \text{ in iteration } t \\ 0 & \text{otherwise} \end{cases} . \qquad (5)$$

   Equation (4) implements a decay of the pheromone level from the previous iteration (evaporation) and an enforcement of the weight that is proportional to maximum matching weights of all paths containing the edge (r, s). The evaporation is controlled by the parameter ρ.

3. Continue with the next iteration $t + 1$ at step 1, until the desired number of iterations is reached.

The best path found in all iterations represents the locally optimal solution to the maximum edge matching problem. Furthermore in the Max–Min Ant System it is differentiated between the globally best solution M_{gb} and the best solution found in a single iteration $M_{ib}(t)$.

The local heuristic weight η is used to improve the ants' behavior. There is no local information that depends only on the edge $(r, s) \in E_B$ independently of the context. Therefore the matching weight developed in Section 3.3 is used. For every possible edge leaving the current matching M of an ant, the heuristic weight for a matching node (r, s) is calculated as the number of matching edges added in the context of the already chosen matching:

$$\eta_{r,s} = w_{M'} - w_M + \eta_{off} = \Delta w_M^{r,s} + \eta_{off} \quad \text{with } M' = \{M, (r, s)\}.$$

The value of $\eta_{r,s}$ resembles the local improvement if a particular edge $(r, s) \in E_B$ is included in M. E.g. in Figure 3 the edges are labeled with $\Delta w_M^{r,s}$ for further illustration. A minor drawback of this method is that during path construction, $\eta_{r,s}$ must be calculated for every possible edge since it is not known in advance, as in e.g. TSP related problems [13]. The parameter η_{off} is an constant weight offset to ensure that $\eta_{r,s} > 0$.

5 Experimental Results

In this section the behavior and the performance of the described search algorithms are illustrated. The advantage of the local heuristics is also justified by the experiments.

Two examples were used for the analysis of the algorithms. The first example (A) consists of two random input graphs. Each graph contains 12 nodes of the same node class and 40 random edges. Besides this a second example (B) that is closely related to hardware reconfiguration was used. The two input graphs are derived from two netlists implementing a 4Bit adder and a 4Bit subtract circuit synthesized for the VirtexII-IPro architecture. Both circuits have a very similar structure. The instances in the input netlists are represented by nodes of different classes. The number of instances that belong to a class of nodes is given for both examples in Table 2(a). The examples were chosen because they allow a comparison between the ACO and the Truncated Search method.

5.1 ACO Setup

The parameters for the ACO search algorithm were chosen to improve the overall performance for the edge matching problem. The number of ants in each iteration was equal to the number of nodes in G_1. The number of iterations is a compromise between runtime and quality of results, especially for large problems where the global optimum is usually not found. The exponents to weight the pheromone level and the local heuristic are best set with $\alpha = \beta = 1$. We found a good value for $\eta_{off} = 1$. The pheromone decay was tuned to $\rho = 0.3$. The extensions of the Max–Min–Ant System were used according to the recommendations in [13]. The pheromone levels were initialized to τ_{max} and were updated using the global best matching M_{gb} every three iterations.

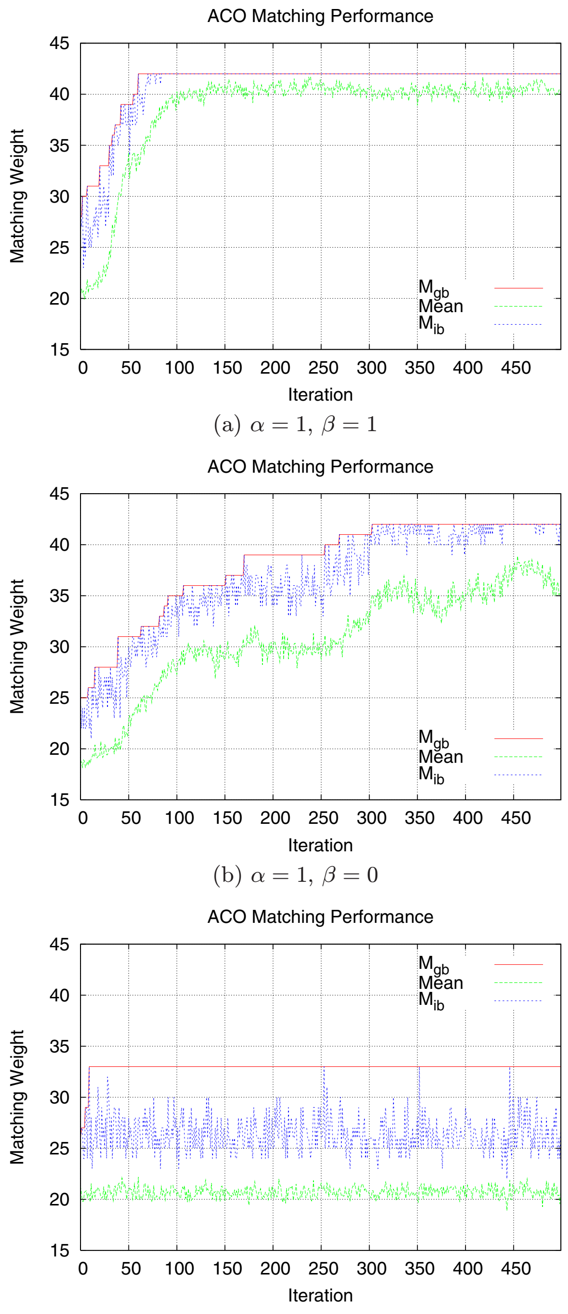
5.2 Swarm Intelligence vs. Randomized Greedy Algorithm

By using different settings for α and β it is possible to tune the ACO behavior to a randomized heuristic or to have "blind" ants that have no local heuristic information at all.

If $\alpha = 0$ then the algorithm has no knowledge about previously constructed matchings. The matchings are chosen according to the local quality of a vertex. A higher local heuristic value improves the probability to choose a certain vertex. The resulting matchings are random and the probability to achieve good results increases with the number of iterations. In theory, setting $\alpha = 0$ and $\beta = \infty$ would result in a greedy heuristic.

Setting $\beta = 0$ means the local heuristic does not effect the probability for an ant to choose a vertex, hence heuristic information is not used. This sets the ACO algorithm to a "blind" behavior, the ants will only learn good matchings by exploration. As the results show, the convergence of the ACO towards the local optimum is much decreased and the quality of results is lower than with the proposed setup.

The experimental results for different settings of α and β are illustrated in Figure 4. The recommended setup in Figure 4(a) clearly shows the best performance of all. There are only few iterations required to find the optimum matching weight. In this example the algorithm converges in the average solution towards the optimum too. The ACO without the local heuristic (Figure 4(b)) does also reach the desired optimum, but requires almost the sixfold number of iterations. As a result we can conclude that the provided local heuristic improves the performance in this ACO algorithm. In contrary, the local heuristic on its own does not provide a good basis for a probabilistic greedy search, as it can be seen in Figure 4(c). In fact, the iteration best matching weight is always in the range of the initial solution of the two parameter settings from Figure 4(a,b).



(a) $\alpha = 1$, $\beta = 1$

(b) $\alpha = 1$, $\beta = 0$

(c) $\alpha = 0$, $\beta = 1$

Figure 4. Performance of the ACO algorithm for example (B) with different settings for α and β. M_{gb} is the best matching weight found so far, M_{it} the best matching weight for this iteration and $Mean$ denotes the mean matching weight in this iteration.

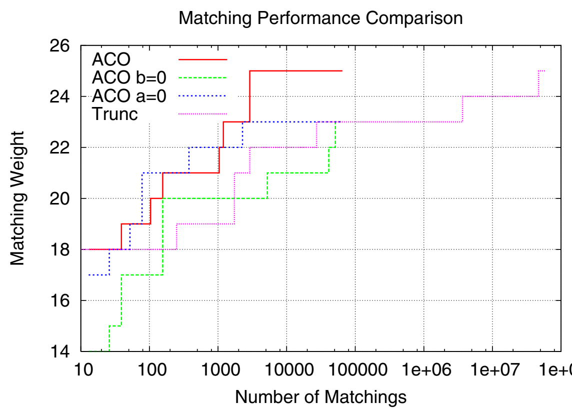


Figure 5. Example (A) Random: Performance of different settings for the matching algorithm.

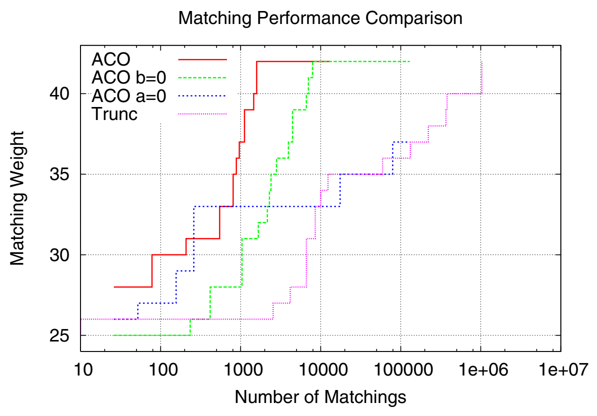


Figure 6. Example (B) Adder/Subtract: Performance of different settings for the matching algorithm.

5.3 Runtime Behavior

A major difference between the search algorithms is in the runtime behavior. While the runtime of the truncated search depends only on the layout of the search tree, the ACO algorithm can be adjusted by setting the number of iterations and ants that solve the problem to the users needs. Allowing more computational effort will improve the solution towards the maximum matching weight. However it is quite impossible for large problems to find the optimal result with the truncated search method, because of the large computational complexity. As an illustration of the required effort, the number of different calculated matchings was recorded for both types of algorithms, see Table 2(b). It can be seen that the truncated search already prunes a lot of obviously bad matchings from the search tree. E.g. in example (B) only $\frac{1}{840000}$ of all possible maximum matchings are searched for the optimal solution. The ACO algorithm requires even less search steps to find the optimum matching. The results are compared for both examples in Figure 5 and 6. In order to compare the behavior of both algorithms, the advances towards the optimum are plotted against the number of calculated matchings. In addition, the runtime behavior of the investigated parameter settings $\alpha = 1$, $\beta = 0$ and $\alpha = 0$, $\beta = 1$ are also included. It appears that the advantages of the ACO and the truncated search are more prominent in example (B), which is more regular structured than example (A).

However, the effort to evaluate a single matching with both algorithms is very different. The truncated search strictly follows the search tree described in Section 3.3, which means that many different evaluated matchings contain a common matching, making the processing very efficient. The ACO algorithm instead uses a different exploration of the search tree for every ant. As a result we observed that the construction of a single matching requires several orders of magnitude more time than the average construction time of a matching with the truncated search method, see Table 3. It is still true that the ACO algorithm requires much less computational effort for most problems due to the very tight search space exploration.

Example	Trunc. Search		ACO	
	per matching	total	per matching	total
(A)	$2.4 \cdot 10^{-6} s$	278s	$3.2 \cdot 10^{-4} s$	0.92s
(B)	$9.6 \cdot 10^{-7} s$	1s	$2.6 \cdot 10^{-4}$s	0.41s

Table 3. Average matching construction times and total runtimes for both examples and algorithms.

We implemented example (B) with the Xilinx ISE tools. Design constraints ensured that matching instances in both netlists occupied identical physical resources. Due to the mapping process some routes where implemented in the logic resources itself. The *Guide Mode* of the place and route tools configured 13 nets in each task identically. Thus, only a very small number (1 and 2 nets in task 1 and 2, respectively) of nets have to be reonfigured between both tasks. Our tools automatically identified the structural similarities and generated the design constraints for the FPGA tools.

	Ex. A		Ex. B	
	G_1	G_2	G_1	G_2
$\mid N_x \cap S_1 \mid$	12	12	1	1
$\mid N_x \cap S_2 \mid$			3	5
$\mid N_x \cap S_3 \mid$			5	5
$\mid N_x \cap S_4 \mid$			4	5
$\mid N_x \cap S_5 \mid$			4	4
$\mid N_x \cap S_6 \mid$			8	8
$\mid E_x \mid$	40	40	44	49

(a)

Example	Complexity	# Matchings	
	Eq. (2)	Trunc. S.	ACO*
(A) Random	$4.8 \cdot 10^8$	115 857 881	2899
(B) Adder / Subtract	$8.4 \cdot 10^{11}$	1 036 366	1586

*Average number of matchings analyzed before the optimum was found.

(b)

Table 2. Parameters of the input graphs (a), the associated theoretical problem complexity and the number of different matchings analyzed for the Truncated Search and the ACO algorithm (b).

6 Conclusion

In this paper we proposed a method to identify structural similarities in tasks of a reconfigurable system. We identified the edge matching problem and proposed two algorithms to solve it. The algorithms have been demonstrated on suitable examples. We have shown that the information about structural similarity can guide the implemention tools to avoid unnecessary configuration.

In the future the techniques will be applied to real world examples. Also the method can be used to aid the design of custom reconfigurable computing architectures. The method provides a measure to assess which reconfigurable routing resources must be provided to support the selected tasks in that architecture.

References

[1] D. Aravind and A. Sudarsanam. High level - application analysis techniques & architectures - to explore design possibilities for reduced reconfiguration area overheads in FPGAs executing compute intensive applications. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005.

[2] G. Chen, M. Kandemir, and U. Sezer. Configuration-sensitive process scheduling for fpga-based computing platforms. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10486, Washington, DC, USA, 2004. IEEE Computer Society.

[3] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 642–649, 2001.

[4] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74, 1998.

[5] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *FCCM*, pages 22–38, 2000.

[6] R. Maestre, F. J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: task scheduling and context management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, December 2001.

[7] N. Moreano, E. Borin, C. de Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):969–980, July 2005.

[8] J. Noguera and R. M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 205–210, New York, NY, USA, 2002. ACM Press.

[9] K. P. Raghuraman, H. Wang, and S. Tragoudas. A novel approach to minimizing reconfiguration cost for LUT-based FPGAs. In *VLSI Design, 2005. 18th International Conference on*, pages 673–676, January 2005.

[10] D. Rakhmatov and S. B. K. Vrudhula. Minimizing routing configuration cost in dynamically reconfigurable FPGAs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 1481–1488, April 2001.

[11] M. Rullmann, S. Siegel, and R. Merker. Optimization of reconfiguration overhead by algorithmic transformations and hardware matching. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 151–156. IEEE, April 2005.

[12] N. Shirazi, W. Luk, and P. Cheung. Automating production of run-time reconfigurable designs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 147–156. IEEE Computer Society Press, April 1998.

[13] T. Stützle and H. H. Hoos. MAX-MIN Ant system. *Future Gener. Comput. Syst.*, 16(9):889–914, 2000.

[14] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings in Computers and Digital Techniques*, 152(2):193–207, March 2005.