Compensating Resource Fluctuations by Means of Evolvable Hardware: The Run-Time Reconfigurable Functional Unit Row Classifier Architecture

Paul Kaufmann[1], Kyrre Glette[2],
Marco Platzner[1], and Jim Torresen[2]

[1] University of Paderborn, Department of Computer Science,
Warburger Str. 100, 33098 Paderborn, Germany
{paul.kaufmann, platzner}@upb.de

[2] University of Oslo, Department of Informatics,
P.O. Box 1080 Blindern, 0316 Oslo, Norway,
{kyrrehg, jimtoer}@ifi.uio.no

Abstract. The *evolvable hardware* (EHW) paradigm facilitates the construction of autonomous systems that can adapt to environmental changes and degradation of the computational resources. Extending the EHW principle to architectural adaptation, we study the capability of evolvable hardware classifiers to adapt to intentional run-time fluctuations in the available resources, i.e., chip area, in this work. To that end, we leverage the *Functional Unit Row* (FUR) architecture, a coarse-grained reconfigurable classifier, and apply it to two medical benchmarks, the Pima and Thyroid data sets from the UCI Machine Learning Repository. While quick recovery from architectural changes was already demonstrated for the FUR architecture, in this work we introduce two reconfiguration schemes helping to reduce the magnitude of degradation after reconfiguration.

1 Introduction

Evolvable hardware (EHW) denotes the combination of evolutionary algorithms with reconfigurable hardware technology to construct self-adaptive and self-optimizing hardware systems [1,2]. EHW's principle is the continuous optimization of its function to be able to react instantly to upcoming events. Several applications of EHW have been proposed, of which some have been very successful. Examples include data compression for printers [3], analog filters [4], evolved image filters [5], evolved shapes for space antennas [6], and high performance reconfigurable caches [7].

EHW-type adaptable systems improve their behavior in response to system internal and external stimuli, offering an alternative to classically engineered adaptable systems. While the adaptation to environmental changes represents the main research line within the EHW community, the ability to balance resources dynamically between multiple concurrent applications is still a rather unexplored topic. One the one hand, an EHW module might run as one out of several applications sharing a system's restricted reconfigurable resources. Depending on the current requirements, the system might decide to switch between multiple applications or run them concurrently, albeit with reduced logic footprints and reduced performance. We are interested in scalable EHW modules and architectures that can cope with such changing resource profiles. On the other hand, the ability to deal with fluctuating resources can be used to support the optimization process, for example by assigning more resources when the speed of adaptation is crucial.

In this work we study the capability of evolvable hardware to adapt to intentional run-time fluctuations in the available resources, i.e., chip area. To demonstrate our approach, we leverage the *Functional Unit Row* (FUR) architecture, a scalable and run-time reconfigurable classifier architecture introduced by Glette et al. [8]. We apply the FUR classifier on two medical benchmarks, the Pima and Thyroid data sets from the UCI Machine Learning Repository. While these benchmarks do not benefit from fast processing times, resource-efficient implementations and run-time adaptation of evolvable hardware, we consider them as model applications because they demonstrate nicely FUR's properties as fast recovery time, the ability to reach high accuracy rates using compact configurations and stable accuracy behavior under a wide range of parameters. We first investigate FUR's general performance for these benchmarks before examining classification behavior during architectural reconfigurations. To minimize the impact of architecture scaling, we introduce two reconfiguration techniques. The reconfiguration techniques gather statistical data during training phases and use it to select the basic pattern matching elements to duplicate or remove when changing the architecture size.

The paper is structured as follows: Section 2 presents the FUR architecture for classification tasks, its reconfigurable variant and the applied evolutionary optimization method. Benchmarks together with an overfitting analysis as well as the experiments with the reconfigurable FUR architecture are shown in Section 3. Section 4 concludes the paper and gives an outlook on future work.

2 The Functional Unit Row Architecture

The Functional Unit Row architecture for classification tasks was first presented by Glette in [8]. It is an architecture tailored to online evolution combined with fast reconfiguration. To facilitate online evolution, the classifier architecture is implemented as a circuit whose behavior and connections can be controlled through configuration registers, similar to the approach of Sekanina [9]. By writing the genome bitstream produced by a *genetic algorithm* (GA) to these registers, one obtains the phenotype circuit which can then be evaluated. In [10], it was shown that the partial reconfiguration capabilities of FPGAs can be used to change the architecture's footprint. The amenability of FUR to partial reconfiguration is an important precondition for our work. In the following, we present the organization of the FUR architecture, the principle of the reconfigurable FUR architecture, and the applied evolutionary technique. For details about the implementation of FUR we refer to [8,11].

2.1 Architecture Overview

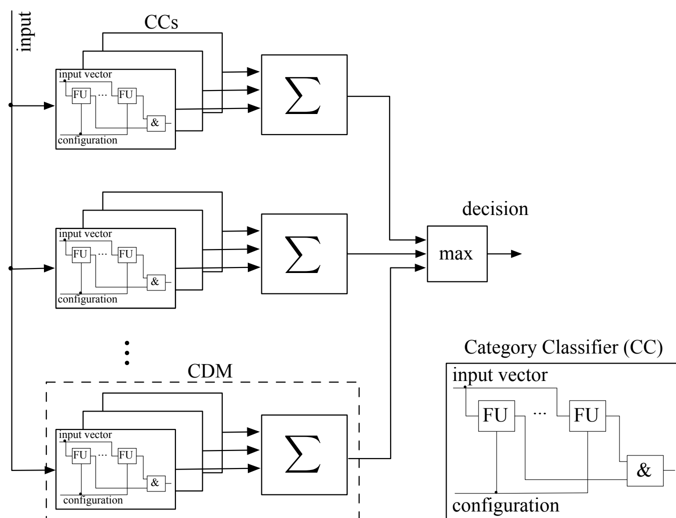

Fig. 1: The Functional Unit Row (FUR) Architecture is hierarchically partitioned for every category into *Category Detection Modules* (CDM). For an input vector, a CDM calculates the likeliness for a previously trained category by summing up positive answers from basic pattern matching elements: the *Category Classifiers* (CC). The CDM with most activated CCs defines the FUR's decision.

Fig. 1 shows the overall organization of the FUR architecture. The overall architecture is rather generic and can be used with different basic pattern matching primitives [12,13]. It combines multiple pattern matching elements into a single module with graded output detecting one specific category. A majority voter decides for a specific category by identifying the module with the highest number of activated pattern matching elements. More specifically, for C categories the FUR architecture consists of C *Category Detection Modules* (CDM). A majority vote on the outputs of the CDMs defines the FUR architecture decision. In case of a tie, the CDM with the lower index wins. Each CDM contains M *Category Classifiers* (CC), basic pattern matching elements evolved from different randomly initialized configurations and trained to detect CDM's category. A CDM counts the number of activated CCs for a given input vector, thus the CDM output varies between 0 and M.

The architecture becomes specific with the implementation of the CCs. In our case we define a single CC as a row of *Functional Units* (FU), shown in Fig. 2. The FU outputs are connected to an AND gate such that in order for a CC to be activated all FU outputs have to be 1. Each CC is evolved from an initial random bitstream, which ensures a variation in the evolved CCs. The number of CCs defines the resolution of the corresponding CDM.

The FUs are reconfigurable by writing the architecture's register elements. As depicted in Fig. 3, each FU behavior is controlled by configuration lines connected to the configuration registers. Each FU has all input bits to the system available at its inputs, but only one data element (e.g., one byte) is selected. This data is then fed to the available functions. While any number and type of functions could be imagined, Fig. 3 illustrates only two functions for clarity. In addition, the unit is configured with a constant value, c. This value and the input data element are used by the function to compute the output of the unit. Based on the data elements of the input, the functions available to the FU elements are *greater than* and *less than or equal*. These functions have by experimentation shown to work well. Altogether, the FU transfer function for an input data a, a constant c and the

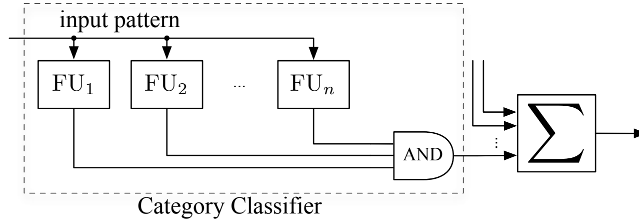

Fig. 2: *Category Classifier* (CC): n *Functional Units* (FU) are connected to an n-input AND gate. Multiple CCs with a subsequent counter for activated CCs define a CDM.

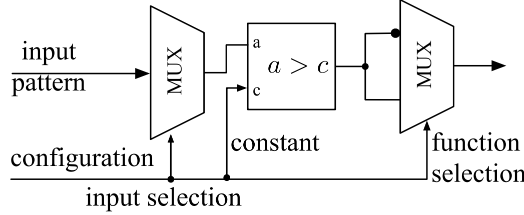

Fig. 3: Functional Unit (FU): The data MUX selects which of the input data to feed to the functions "$>$" and "$\leq$". The constant c is given by the configuration lines. Finally, a result MUX selects which of the function results to output.

function selection bit s is defined as:

$$\mathrm{FU}(a, c, s) = \begin{cases} s & : \text{ if } a > c \\ \bar{s} & : \text{ else.} \end{cases}$$

In this specific FU configuration, the FUR classification principle is closely related to the classification principle of decision trees, which realize decision boundaries with sections of straight lines that must be parallel to the axes of the input space spanned by all input data elements.

2.2 The Reconfigurable FUR Architecture

During the design time, FUR's architecture can be parametrized along three dimensions, namely the number of

- categories,
- CCs in a category, and
- FUs in a CC.

The authors of the FUR architecture show in [10] that the partial reconfiguration capabilities of FPGAs can be used to change the architecture's footprint dynamically. For our experiments, we decided to vary the number of CCs in a CDM for the following reasons: The number of categories is typically known a priori and is fixed. When comparing the classification principles of the FUR architecture and decision trees, the number of FUs in a CC can be seen as analog to decision trees' depth, which roughly represents the dimensionality of the decision space. This is highly application specific. Reducing the amount of FUs per CC without increasing the number of CCs in a CDM would more likely create systems fundamentally unable to reach the high classification rates of a proper configured FUR architecture.

Additional motivation for changing the number of CCs in a CDM is that the FUR architecture is fully operational with only one CC per CDM. The number of CCs in a CDM can be seen as the CDM's resolution. While the FUR architecture shows basic discrimination abilities with one or few CCs in a CDM, increasing the number of CCs typically makes the accuracy rate reach higher levels.

Reconfiguration of the FUR architecture is sketched in Fig. 4. For a sequence $I = \{i_1, i_2, \ldots, i_k\}$ we evolve a FUR architecture having i_1 FUs per CDM, then switching to i_2 FUs per CDM and re-evolving the architecture without flushing the configuration evolved so far.

In this investigation we want to examine the sensitivity of the classification accuracy to the changes described above, and how fast the evolutionary algorithm is able to reestablish pre-reconfiguration accuracy rates. Furthermore, we would like to investigate if our strategies for replacing and duplicating the "best" and "worst" CCs can reduce the impact of architectural reconfigurations on the accuracy rate.

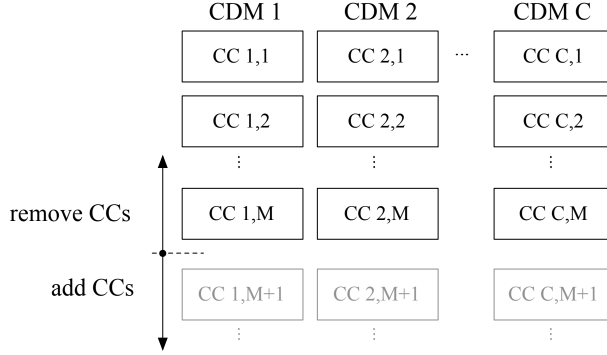

Fig. 4: Reconfigurable FUR Architecture: The FUR architecture is parametrized by the number of categories, category classifiers (CC) and functional units (FU) per CC. While the number of categories is fixed and the number of FUs is largely application dependent, we scale the FUR architecture by changing the number of category classifiers in a category detection module.

2.3 Evolution

To evolve a FUR classifier, we employ a $1 + 4$ *Evolutionary Strategy* (ES) scheme variation introduced by Miller and Thomson in [14]. In this particular scheme one parent creates four off-spring individuals. The parent propagates to the new generation only if all off-spring individuals demonstrate lower accuracy rates. Otherwise, the best offspring individual becomes the new parent. The scheme is illustrated in Alg. 1. The fitness of a candidate solution is evaluated as its classification accuracy. New individuals are derived from the parent by mutating three genes in every CC. In contrast to previous work [8], we do not use incremental evolution but evolve the complete FUR architecture in a single ES run.

Algorithm 1: `ES-step` (p,λ) - execute one step of a 1+λ Evolutionary Strategy scheme.

```
  Input: individual p, reproduction parameter λ
  Output: off-spring individual p
1 for i = 1, . . . , λ do
2 |   p' ← mutate(p)
3 |   if p not better than p' then
4 |   |   p ← p'
5 |   end
6 end
```

FUR's functional granularity level covers basic arithmetic functions operating on binary encoded numbers. Therefore, all elements of an input vector are converted to 8-bit binary encoded numbers. With n elements in a single input vector, the configuration bitstring for a single FU amounts for $n_u = \lceil \log_2(n) \rceil + 8 + 1$ bits. Having c categories, n_f FUs in a CC as well as n_c CCs in a CDM, the total genotype is $c \cdot n_c \cdot n_f \cdot n_u$ bits long.

3 Experiments and Results

This section presents two kinds of results. First, we analyze the general behavior of FUR for the Pima and Thyroid data sets by successively testing a range of architecture parameter combinations. Combined with an overfitting analysis, we are then able to picture FUR's general behavior for these benchmarks. In the next section, we select a good-performing configuration to investigate FUR's performance, when being reconfigured during run-time.

3.1 Benchmarks

For our investigations we rely on the UCI machine learning repository [15] and specifically, on the Pima and the Thyroid benchmarks. Pima, or the *Pima Indians Diabetes* data set is collected by the John Hopkins University in Baltimore, MD, USA and consists of 768 samples with eight feature values each, divided into a class of 500 samples representing negative tested individuals and a class of 268 samples representing positive tested individuals. The data of the Thyroid benchmark represents samples

of regular individuals and individuals suffering hypo- and hyperthyroidism. Thus, the samples are divided into 6.666, 166 and 368 samples representing regular, subnormal and hyper-function individuals. A sample consists of 22 feature values. Assuming a FUR classifier with 4 FUs in a CC and 10 CCs in a CDM, the genotype lengths amount for $2 \cdot 10 \cdot 4 \cdot (\lceil \log_2(8) \rceil + 8 + 1) = 960$ bits for the Pima and $3 \cdot 10 \cdot 4 \cdot (\lceil \log_2(22) \rceil + 8 + 1) = 1680$ bits for the Thyroid data sets, respectively.

Both benchmarks do not rely on high classification speeds and run-time adaptation of EHW hardware classifiers, however, these benchmarks have been selected because of their pronounced effects in the run-time reconfiguration experiments presented in the next section revealing FUR's characteristics.

3.2 Accuracy and Overfitting Analysis

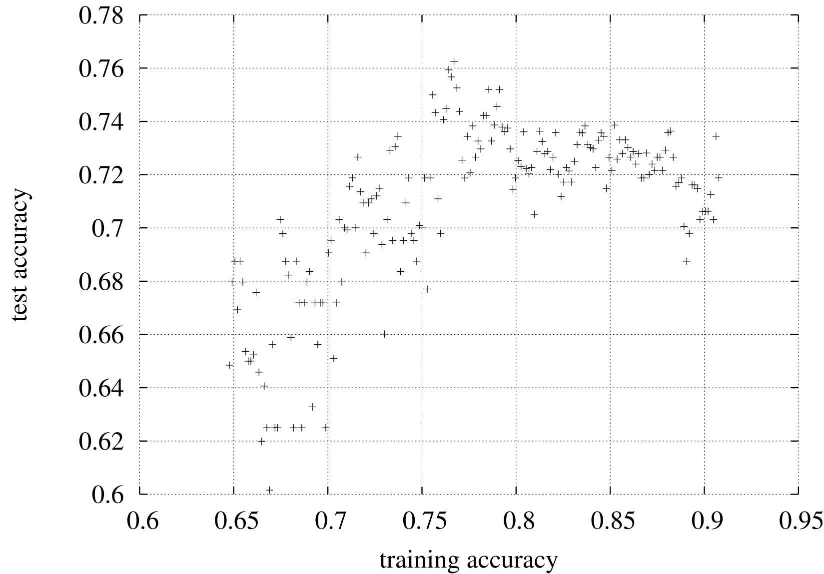

Fig. 5: Overfitting analysis for the Pima data set and FUR architecture having 8 FUs per CC and 30 CCs per CDM. In this example, the termination training accuracy lies around 0.76, before the test accuracy begins to decline, indicating overfitting.

We implement parameter analysis of the FUR architecture by a grid search over the number of CCs and number of FUs. For a single (i, j)-tuple, where i denotes the number of CCs and j the number of FUs, we evolve a FUR classifier by running $1 + 4$ evolutionary strategies for 100.000 generations. In contrast to our original work on the FUR architecture in [16], we do not use incremental evolution but evolve the complete FUR architecture in a single ES run. Thereby we use a mutation operator modifying three genes in every CC. As we employ a 12-fold cross validation scheme, the evolution is repeated 12 times while alternating the training and test data sets. We select $n = 12$ in our experiments as an acceptable compromise between computational complexity and precision of classifier evaluation. During the evolution, we log for every increase in the training accuracy FUR's performance on the test data set. The test accuracies are not used while the evolution runs. To detect the tipping point where FUR starts to overfit, i.e. where FUR learns to match each training vector instead of learning the general model, we average the test accuracies logged during the evolutionary runs and select the termination training accuracy according to the highest average test accuracy. With an *a priori* known termination training accuracy we can evolve a new classifier until it reaches the termination training accuracy and expect it having on average a good classification performance for unknown data. An example is shown in Fig. 5 for the Pima benchmark and the $(30, 8)$ configuration. The test average accuracy, drawn along the y-axis, rises in relation to the average training accuracy, drawn along the x-axis, until the training accuracy reaches 0.76. After this point the test accuracy degrades gradually. Consequently, we note 0.76 and 0.76 as the best combination of test and termination training accuracies.

To examine the general FUR performance for the Pima and Thyroid data sets, we configure and evaluate the FUR architecture for all combinations of $2, 4, 6, \ldots, 20$ FUs per CC and for 2, 4, 6, 8, 10, 14, 16, 20, 25, 30, 35, 40, 50, 60, 70, 80 CCs. Fig. 6 displays the results. In the horizontal level the diagrams span the parameter area of CCs and FUs. The accuracy for each parameter tuple is drawn along the z-axis with a projection of equipotential accuracy lines on the horizontal level. While the test accuracies for the Pima benchmark, presented in Fig. 6(a) are largely independent from the number of FUs and CCs with small islands of improved behavior around the $(8, 8-10)$ configurations, the Thyroid benchmark presented in Fig. 6(c) has a performance loss in regions with a large number of FUs and few CCs.

Tables 1 and 2 compare FUR's results for the Pima and the Thyroid benchmarks to related work. We use additionally the data mining tool RapidMiner [17] to create numbers for standard and state-of-the-art algorithms and their modern implementations.

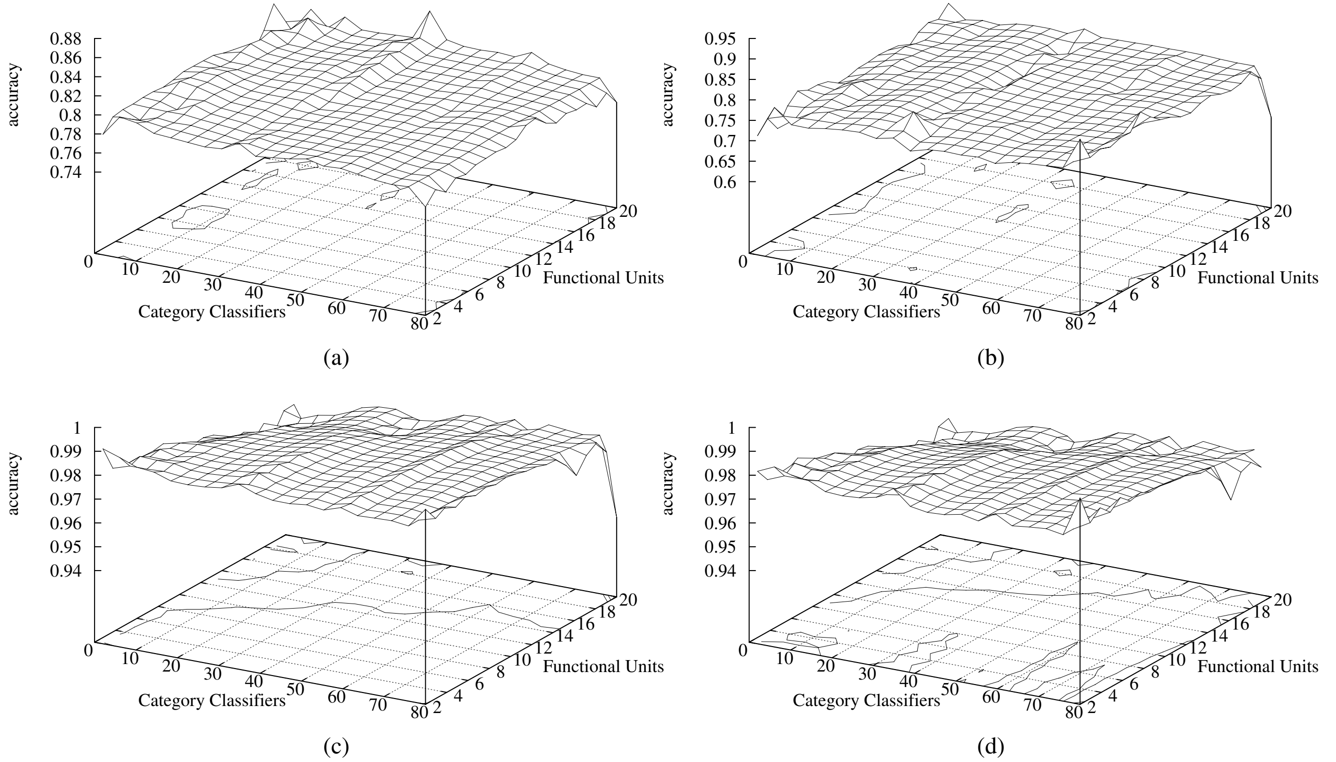

Fig. 6: Pima and Thyroid overfitting analysis: Best generalization and the according termination training accuracies for the Pima (a) (b) and the Thyroid (c) (d) benchmarks, respectively.

The following algorithms are evaluated with 12-fold cross validation: *Decision Trees* (DTs), k-th *Nearest Neighbor* (kNN), *Multi-layer Perceptrons* (MLPs), *Linear Discriminant Analysis* (LDA), *Support Vector Machines* (SVMs) and *Classification and Regression Trees* (CART). For the Pima benchmark the FUR architecture demonstrated higher recognition rates than any other method. It forms together with SVMs, LDA, Shared Kernel Models and kNNs a group of best performing algorithms within a 3% margin. The accuracy range of the Thyroid-benchmark is much smaller because of the disproportional category data sizes and a single dominant category amounting for 92.5% of the data. In this benchmark the FUR architecture lies 0.66% behind the best algorithm.

3.3 Reconfiguration Schemes for Functional Unit Row Architecture

In this section we investigate FUR's run-time adaptation capabilities to gradual and radical changes in resource sizes. To this end, we configure FUR with 4 FUs per CC and change the number of CCs every 50.000 generations. We split the data set into disjoint training and test sets similar to the previously used 12-fold cross validation scheme and start the training with 10 CCs. Then, we gradually change the number of CCs to 9, 8,. . . , 1 and back to 2, 3,. . . , 10 executing altogether 10^6 generations. In the second experiment we investigate larger changes, switching from 10 to 4 to 2 to 5 and back to 10 CCs. For sound results, we repeat the experiments 96 and 32 times for first and second experiments, respectively.

Our basic implementation of FUR's reconfiguration reduces and increases the amount of CCs in a CDM by removing randomly selected and adding randomly initialized CCs to a CDM. In order to improve adaptation times during architectural reconfigurations, we define two additional schemes to change the amount of CCs in a CDM by removing and duplicating "worst" and "best" CCs, respectively. To quantify the quality of a CC, we define for every CC a penalty counter that is increased by the number of wrongly activated CCs in the same CDM for some input vector. A specific CC's counter is only increased when the CC itself decides incorrectly. The rationale behind this is that FUR's global decision is taken at the CDM level. Thus, a CDM with, for instance, 4 wrongly decided CCs is more likely to cause an incorrect global decision than a CDM with only 2 wrongly decided CCs. In the first case, every CC in the particular CDM with an incorrect decision adds a 4 to its penalty counter while in the second case, a 2 is added. Consequently, a CC is considered "bad" when having a higher penalty counter and "good" otherwise.

Table 1: Pima benchmark: Error rates and standard deviation (SD) in %. We use the data mining toolbox RapidMiner [17] to evaluate the algorithms marked by "*". Preliminary, we identify good performing algorithm parameters by a grid search. Remaining results are taken from [18].

Algorithm	Error Rate	± SD
FUR	**21.35**	
SVM*	22.79	4.84
LDA*	23.18	4.64
Shared Kernel Models	23.27	2.56
kNN*	23.56	3.07
GP with OS, |pop|=1.000	24.47	3.69
CART*	25.00	3.61
DT*	25.13	4.30
GP with OS, |pop|=100	25.13	4.95
MLP*	25.26	4.50
Enhanced GP	25.80 – 24.20	
Simple GP	26.30	
ANN	26.41 – 22.59	1.91 – 2.26
EP / kNN	27.10	
Enhanced GP (Eggermont et al.)	27.70 – 25.90	
GP	27.85 – 23.09	1.29 – 1.49
GA / kNN	29.60	
GP (de Falco et al.)	30.36 – 24.84	0.29 – 1.30
Bayes	33.40	

Gradually Changing the FUR Size: Fig. 7 compares accuracy drop magnitudes during architectural reconfiguration using "random", "best" and "worst" schemes. The diagrams plot averaged numbers over 96 experiments for the Pima data set. The top diagrams show exemplarily the training behavior when removing a randomly CC and adding a randomly initialized CC to FUR architecture. Diagrams in the next three lines illustrate test accuracy behavior for the "random", "best" and "worst" reconfiguration schemes. An obvious conclusion is that removing "worst" CCs result in smallest accuracy drops. Analog, duplicating "best" CCs helps minimizing accuracy drops when adding resources to the FUR architecture. Generally, we observe for the Pima benchmark the following:

- The training and test accuracies drop for any reconfiguration scheme for almost any positive and negative changes in the number of CCs and recover subsequently. The drops are slightly larger for configurations with few CCs. The average accuracy drops, as summarized in Tab. 3, are minimized when removing "worst" and duplicating "best" CCs during FUR reconfiguration. Randomly removing and initializing new CCs comes second while removing "best" and duplicating "worst" CCs result in largest accuracy losses. Altogether, the differences between the reconfiguration schemes are rather small, amounting roughly for up to 3.5% and 1% for removing and adding CCs, respectively.
- We observed maximal training accuracies to be somewhat lower when using fewer CCs, while the test accuracies tend to stay at the same levels. Only for configurations with one and sometimes two CCs per CDM, the accuracy rates did not reach the pre-switch levels. The gaps, however, are small, roughly amounting for up to 5%.
- Test accuracies are recovered quickly for most schemes and FUR configurations. However, the strategy of removing "worst" and "random" CCs maintains a fast recovery speed also when going below five CCs per CDM, whereas removing "best" leads to significantly longer times before the asymptotic test accuracies are recovered.
- The test accuracies are mostly located between 0.72 and 0.76, independent of the changes in the number of CCs. Thus, and this is the main observation, the FUR architecture shows to a large extent a robust test accuracy behavior under reconfiguration for the Pima benchmark.

For the Thyroid benchmark we can make the following conclusions:

- The test accuracies drop significantly when changing the number of CCs. The drops are roughly three to five times larger reaching up to 40%. Reducing FUR size, the accuracy drops became smallest when removing "worst" CCs during reconfigurations, followed by schemes selecting random and "best" CCs. When increasing the number of CCs, the smallest accuracy drops are obtained from the random CC duplication scheme, followed by schemes duplicating "best" and "worst" CCs. The difference between the random and "best" duplication scheme is, however, small accounting for 0.69%.
- As anticipated by previous results shown in Fig. 6 (c), the test accuracy degrades for FUR architecture configurations with very few CCs. For instance, a FUR configuration with only one CC demonstrates an error rate of 7%. This is considerably low as the baseline recognition rate lies at 92.5%, which corresponds to an error rate of 7.5% due to a single category amounting for 6.666 out of 7200 vectors.

Table 2: Thyroid benchmark: Error rates and standard deviation (SD) in %. We use the data mining toolbox RapidMiner [17] to evaluate the algorithms marked by "*". Preliminary, we identify good performing algorithm parameters by a grid search. Remaining results are taken from [18].

Algorithm	Error Rate	± SD
DT*	0.29	0.18
CART*	0.42	0.27
CART	0.64	
PVM	0.67	
Logical Rules	0.70	
FUR	**1.03**	
GP with OS	1.24	
GP	1.44 – 0.89	
BP + local adapt. rates	1.50	
ANN	1.52	
BP + genetic opt.	1.60	
GP	1.60 – 0.73	
Quickprop	1.70	
RPROP	2.00	
GP (Gathercole et al.)	2.29 – 1.36	
SVM*	2.35	0.51
MLP*	2.38	0.62
ANN	2.38 – 1.81	
PGPC	2.74	
GP (Brameier et al.)	5.10 – 1.80	
kNN*	5.96	0.44

Table 3: Averaged accuracy drops in % over 96 algorithm runs. $1+4$ ES is executed for 50.000 generations between the reconfigurations. During a reconfiguration randomly selected, "best" or "worst" CCs are removed or duplicated. Bold numbers indicate best-performing replacement strategy.

		$10 \rightarrow 9 \rightarrow \cdots \rightarrow 1$		$1 \rightarrow 2 \rightarrow \cdots \rightarrow 10$	
		training	test	training	test
Pima	random	10.87	5.70	8.33	5.70
	low penalty	13.57	7.90	7.18	**5.15**
	high penalty	9.39	**4.23**	8.90	6.11
Thyroid	random	23.94	23.77	15.91	**15.74**
	low penalty	40.87	40.73	16.13	16.03
	high penalty	12.21	**12.00**	20.60	20.53

– The main result is that given enough resources reconfigurations are quickly compensated. The limitation in the case of the Thyroid benchmark is a minimum amount of CCs required to achieve high recognition rates.

Radically Changing FUR's Size: Challenges of an autonomous and adaptable embedded system are manifold. It may for instance have to react to aperiodic events where the solution may benefit from additional resources. With typically restricted energy the embedded system faces sometimes also the challenge of balancing resources between multiple applications. We anticipate resource assignment changes in such situations being of a more radical nature. To investigate FUR's behavior in such situations, we define a second reconfiguration sequence switching form 10 to 4 to 2 to 5 and back to 10 CCs. In each reconfiguration step we nearly halve or double FUR's size. The results are summarized in Tab. 4. Almost all observations and conclusions made in the previous experiment hold also for the current experiment. The important differences are:

– The accuracy drops are now mostly two to three times as large compared to gradual FUR size changes. Despite the dramatic numbers, accuracies recover similarly fast.
– Analog to the previous experiment, removing "worst" CCs and duplicating "best" CCs reduces the accuracy drops for Pima and Thyroid benchmarks. There is, however, one exception. Lowest test accuracy drops when switching from 2 to 5 CCs in the Thyroid benchmark are achieved by duplicating the "worst" CC three times.

In summary, for all experiments in this section we can conclude that the FUR architecture is exceptionally fast in recovering from architectural reconfigurations, given enough resources are provided for learning. Still, the proposed schemes of removing "worst"

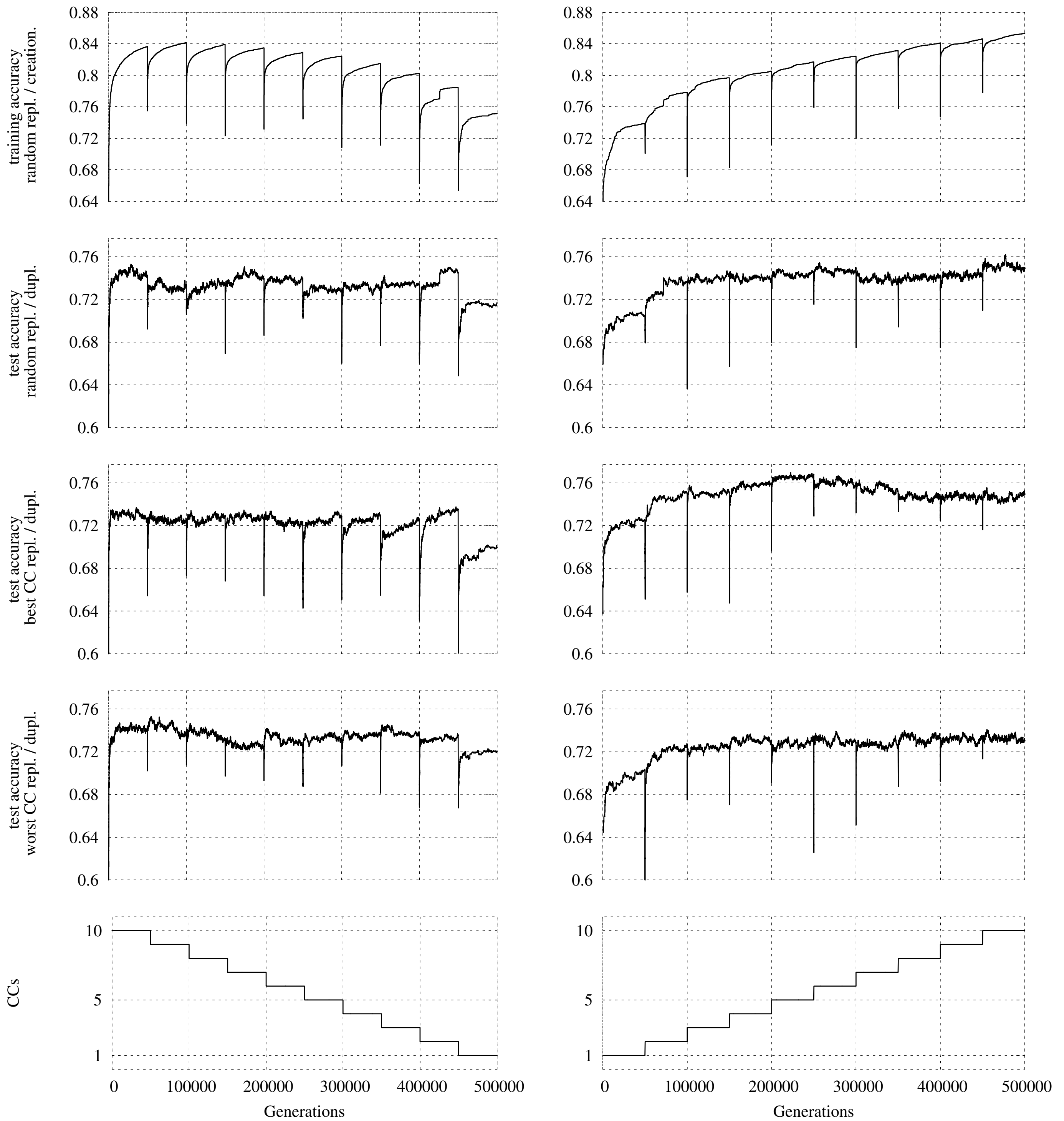

Fig. 7: Gradually reconfiguring the FUR architecture using the Pima data set. The left column presents FUR's averaged classification behavior in % when reducing and the right column when increasing the number of CCs. The diagrams on the bottom show the number of CCs in the system. Both diagrams on the top show exemplarily the training behavior for the randomized reconfiguration scheme. Diagrams in the second, third and fourth lines show test behavior for the randomized, "best", and "worst" FUR reconfiguration schemes, respectively.

and adding "best" CCs help to reduce the impact on the classification rate after reconfiguration of the architecture dimensions. This is both in terms of lower magnitudes on the instantaneous accuracy drops, as well as a shortened recovery time before pre-reconfiguration test accuracies are regained.

Table 4: Averaged accuracy drops in % over 32 algorithm runs. $1 + 4$ ES is executed for 50.000 generations between the reconfigurations. During a reconfiguration randomly selected, “best” or “worst” CCs are removed or duplicated. Bold numbers indicate best-performing replacement strategy.

		$10 \rightarrow 4$		$4 \rightarrow 2$		$2 \rightarrow 5$		$5 \rightarrow 10$	
		training	test	training	test	training	test	training	test
Pima	random	21.90	13.76	17.75	9.91	13.46	11.86	18.27	15.42
	low penalty	21.59	10.93	19.94	11.52	8.98	**6.34**	16.52	**9.32**
	high penalty	16.65	**10.30**	10.57	**5.61**	14.66	11.91	21.71	16.35
Thyroid	random	60.00	59.37	45.96	45.55	30.29	30.27	44.13	44.00
	low penalty	54.50	54.28	54.75	54.72	30.14	29.88	34.90	**34.93**
	high penalty	34.27	**33.91**	35.71	**35.89**	18.65	**18.30**	71.84	72.16

4 Conclusion

In this work we propose to leverage the FUR classifier architecture for creating evolvable hardware systems that can cope with fluctuating resources. We describe FUR's architecture and experimentally evaluate it on two medical benchmarks. In the first experiment we analyze FUR's overfitting behavior and demonstrate that FUR performs similar or better than conventional state-of-the-art classification algorithms. Then we investigate FUR performance during architectural reconfigurations. This is done by reducing or increasing the available resources and measuring the accuracy behavior during the transitions. To reduce the impact of reconfiguration on the accuracy rate, we also introduce two reconfiguration schemes for adding and removing Functional Unit rows to an existing FUR architecture. We demonstrate that the generalization performance of FUR is very robust to changes in the available resources as long as a certain amount of CCs is present in the system. While for the Pima benchmark the differences between the reconfiguration schemes and also the absolute accuracy drops are small, for the Thyroid benchmark we can substantially improve the behavior of FUR during a reconfiguration.

In future we will continue our work on run-time reconfiguration support for an FPGA-based hardware prototype of the system, and will benchmark the performance with regard to training and reconfiguration time. In addition, we will extend the benchmarking scenarios for resource fluctuations to reflect real-world situations in embedded systems.

References

1. de Garis, H.: Evolvable Hardware: Genetic Programming of a Darwin Machine. In: Intl. Conf. on Artificial Neural Nets and Genetic Algorithms. Springer (1993) 441–449
2. Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H., Furuya, T.: Evolving Hardware with Genetic Learning: a First Step Towards Building a Darwin Machine. In: From Animals to Animats, MIT Press (1993) 417–424
3. Tanaka, M., Sakanashi, H., Salami, M., Iwata, M., Kurita, T., Higuchi, T.: Data compression for digital color electrophotographic printer with evolvable hardware. In Sipper, M., et al., eds.: Intl. Conf. on Evolvable Systems (ICES). Volume 1478 of LNCS. Springer (1998) 106–114
4. Koza, J., Keane, M., Streeter, M.: Routine high-return human-competitive evolvable hardware. NASA/DoD Conference on Evolvable Hardware (2004) 3–17
5. Sekanina, L.: Evolutionary Design Space Exploration for Median Circuits. In: Applications of Evolutionary Computing. Volume 3005 of LNCS., Springer (2004) 240–249
6. Lohn, J., Hornby, G., Linden, D.: Evolutionary antenna design for a NASA spacecraft. In O'Reilly, U.M., Yu, T., Riolo, R.L., Worzel, B., eds.: Genetic Programming Theory and Practice II. Springer (2004) 301–315
7. Kaufmann, P., Plessl, C., Platzner, M.: EvoCaches: Application-specific Adaptation of Cache Mappings. In: Adaptive Hardware and Systems (AHS), IEEE CS (2009) 11–18
8. Glette, K., Torresen, J., Yasunaga, M.: An Online EHW Pattern Recognition System Applied to Face Image Recognition. In: Applications of Evolutionary Computing (EvoWorkshops). Volume 4448 of LNCS. Springer (2007) 271–280
9. Sekanina, L., Ruzicka, R.: Design of the Special Fast Reconfigurable Chip Using Common FPGA. In: Design and Diagnostics of Electronic Circuits and Systems (DDECS). (2000) 161–168
10. Torresen, J., Senland, G., Glette, K.: Partial Reconfiguration Applied in an On-line Evolvable Pattern Recognition System. In: NORCHIP 2008, IEEE (2008) 61–64
11. Glette, K., Torresen, J., Yasunaga, M.: Online Evolution for a High-Speed Image Recognition System Implemented On a Virtex-II Pro FPGA. In: Adaptive Hardware and Systems (AHS), IEEE (2007) 463–470
12. Glette, K., Gruber, T., Kaufmann, P., Torresen, J., Sick, B., Platzner, M.: Comparing Evolvable Hardware to Conventional Classifiers for Electromyographic Prosthetic Hand Control. In: Adaptive Hardware and Systems (AHS), IEEE (2008) 32–39
13. Yasunaga, M., Nakamura, T., Yoshihara, I.: Evolvable Sonar Spectrum Discrimination Chip Designed by Genetic Algorithm. In: Systems, Man and Cybernetics. Volume 5., IEEE (1999) 585–590
14. Miller, J., Thomson, P.: Cartesian Genetic Programming, Springer (2000) 121–132

15. Asuncion, A., Newman, D.: UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences (2007)
16. Glette, K., Torresen, J., Yasunaga, M., Yamaguchi, Y.: On-Chip Evolution Using a Soft Processor Core Applied to Image Recognition. In: Adaptive Hardware and Systems (AHS), IEEE (2006) 373–380
17. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: Rapid Prototyping for Complex Data Mining Tasks. In: Intl. Conf. on Knowledge Discovery and Data Mining (KDD). (2006) 935 – 940
18. Winkler, S.M., Affenzeller, M., Wagner, S.: Using Enhanced Genetic Programming Techniques for Evolving Classifiers in the Context of Medical Diagnosis. In: Genetic Programming and Evolvable Machines. Volume 10(2)., Kluwer Academic Publishers (2009) 111–140