Generic Self-Adaptation to Reduce Design Effort for System-on-Chip

Andreas Bernauer, Wolfgang Rosenstiel
Department of Computer Engineering
Wilhelm-Schickard-Institute for Informatics
University of Tübingen, Germany
Email: {bernauer,rosenstiel}@informatik.uni-tuebingen.de

Oliver Bringmann
Forschungszentrum Informatik
Karlsruhe, Germany
Email: bringmann@fzi.de

Abstract

We investigate a generic self-adaptation method to reduce the design effort for System-on-Chip (SoC). Previous self-adaptation solutions at chip-level use circuitries which have been specially designed for the current problem by hand, leading to an elaborate and inflexible design process, requiring specially trained engineers, and making design reuse difficult. On the other hand, a generic self-adaptation method that can be used for various self-adaptation problems promises to reduce the necessary design effort, but may come with reduced performance and other costs.

In this paper, we analyze the performance, self-adaptation capabilities and costs of a generic self-adaptation method. The proposed method allows chip-level self-adaptation of a SoC, can tolerate unforeseen events, and can generalize from previous self-adaptation tasks. Furthermore, the method helps to improve the design process by allowing design reuse, providing generic applicability, and offering a uniform design process for various self-adaptation tasks. Simulation results show that the performance of our method lies only 10% below the performance of a perfect, non-adaptive system in the average case, and only 32% in the worst case. In case of unforeseen events, where the performance of a non-adaptive system decreases significantly, the method can keep its performance level by self-adaptation. We also compare other costs involved.

1. Introduction

According to the International Technology Roadmap for Semiconductors (ITRS) [1], the number of System-on-Chip (SoC) designs is expected to increase strongly. The major advantages of SoC designs in comparison to other designs are higher levels of system integration and reduced design costs. However, as integration density is increasing and time to market is shrinking, keeping SoC design effort constant turns out to be difficult. To manage the increasing design complexity, the ITRS estimates a requirement of a design reuse rate of 70% until 2015 and 90% until 2020 [1, System Drivers Chapter].

Besides increasing integration densities, increasing transistor variability [2], [3], process variation [4] and degradation effects [5] add to the design complexity, making it increasingly difficult for manufacturers to fulfill the expectations of their customers with respect to the reliability of the products [6]. For the same reasons, traditional worst-case design becomes less favorable for nano-scale chips, as the design margins that are required to cover the worst cases would use most of the benefits of the next technology node [7]. Instead, designers turn to better-than-worst-case design [8], where they design the chip for the typical case and add a special circuit to handle the presumably rare worst cases.

Another way to help better-than-worst-case designs is to add circuits that enable the chip to self-adapt individually to its actual manifestation of variability, variation and degradation. The self-adapting circuits allow the chip to reach its individually maximally possible performance. Essentially, self-adaptation moves some decisions that have formerly been taken at design time to the run time of a chip, as at design time, the particular optimal settings of the chip are unknown. For example, in [9] the authors describe how to tune the clocking of the pipeline stages after chip fabrication to compensate for process variability. In [10], the authors propose a correction circuit that adjusts the supply voltage of an SRAM at run time so that it balances error rate and power consumption. Taking these run-time decisions at chip level leads to short reaction times, and allows the adaptations to be transparent to the operating system.

Although self-adaptation circuits help to realize better-than-worst-case designs, current self-adaptation circuits (such as [9], [10]) are special-purpose and hence increase the design effort. Their design requires specially trained engineers who, faced with a self-adaptation problem, must craft an individual solution while considering all possible events that may affect it, overall prolonging the design process. Furthermore, as the circuits address only one particular problem, they cannot be easily reused for similar or different problems in another design, that is, their design reuse rate is low. Faced with the increased design effort, the designer of nano-scale SoCs is forced to decide between the poor chip performance of a worst-case design and the prolonged design process of a better-than-worst-case design that uses special-purpose self-adaptation circuits.

In this paper, we investigate the feasibility of a generic self-adaptation circuit and method that offers a middle ground between a poor chip performance and a prolonged design process. In contrast to special-purpose self-adaptation systems, the generality of a generic self-adaptation system offers a high design reuse rate, while its self-adaptivity allows better-than-worst-case designs. We identify a possible candidate for

This work is partially funded by Deutsche Forschungsgemeinschaft, Priority Program Organic Computing (SPP 1183) under the grant RO 1030/14-2.

a generic self-adaptation system based on the reinforcement algorithm XCS [11], analyze its general performance, its self-adaptation capabilities and its costs, and discuss how it helps to mitigate the design effort. To the best of our knowledge, this is the first paper investigating a generic self-adaptation methodology at chip-level.

The remainder of this paper is structured as follows: In Section 2 we determine the requirements of a generic self-adaptation system and present our candidate for the evaluator of such a method. Section 3 describes the experimental setup with which we analyze our generic self-adaptation system. Section 4 shows the results of the evaluation in terms of performance, self-adaptation capabilities, and design- and run-time costs. Section 5 concludes.

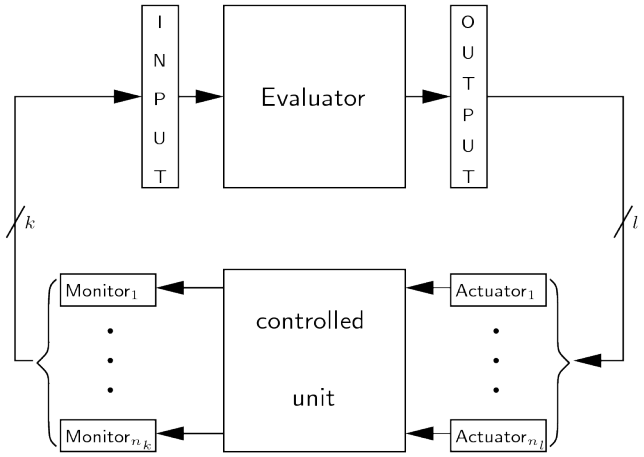

Fig. 1. Generic self-adaptation system

2. Generic self-adaptation at chip-level

In this section, we determine the requirements of a generically applicable self-adaptation system that helps to reduce the design effort, and present our generic self-adaptation method, which is based on the reinforcement algorithm XCS [11].

Naturally, a generic self-adaptation system will provide the usual input and output interface of any other self-adaptation system (compare to Figure 1): a *monitor* with which the self-adaptation system reads in the current state of the controlled unit and an *actuator* that executes the action of the self-adaptation system to move the controlled unit from the current state towards the desired state. To allow generic reusability, the monitor and actuator signals will be actual bit vectors at the input and output of the generic self-adaptation system. They respectively consist of n_k and n_l individual monitors and actuators, providing k and l total bits of monitor and actuator data. At the heart of the self-adaptation system lies the *evaluator*, which reasons about the current state to suggest a preferably optimal action that moves the controlled unit to the desired state. To allow generic reusability, the evaluator has to be as ignorant as possible to the meaning of the input and output signals.

In current special-purpose self-adaptation systems, the evaluator is crafted by the designer and has to be instructed explicitly, which prolongs the design process. To reduce the need of explicit manual instruction, we propose to use an evaluator that automatically learns the optimal actions for a given situation by itself. Of course, as learning involves making errors, this kind of evaluators is only suitable for systems where errors are not fatal or guarding circuits prevent the errors. Usually, the designer knows the desired system states, but does not know which actions lead there from a given system state—one of the reasons that prolongs the design process of special-purpose self-adaptation systems. From the machine learning algorithms, we identify the reinforcement algorithms as the most suitable evaluators, as the designer can easily provide a reward function that favors the desired systems states. In particular, temporal difference (TD) learners [12] provide the advantage that they do not require a model of the environment, its reward, or the probability distribution of the next state, which suits well our goal of a generic self-adaptation method.

However, if the designer knows the best action for a given system state, it is desirable that he or she could tell the generic self-adaptation system so, not only to save investments already taken to solve some aspects of a self-adaptation problem, but also to help the learning process of the evaluator. From the temporal difference learners, the learning classifier systems [13] suits this goal well, as they represent acquired knowledge with human-readable rules. This allows the designer not only to insert his or her expert knowledge in the form of new rules, but also to understand what the evaluator has learned and to transfer knowledge to similar adaptation problems, which further helps to reduce design efforts. This is in contrast to other machine learning algorithms such as neural networks, where the acquired knowledge is stored opaquely in a weight matrix. The learning classifier system XCS [11] is reported to learn accurately and completely [14]. XCS has also been shown to be implementable in hardware so that it allows chip-level control. In [15] the authors present an implementation of the XCS on an FPGA. The authors of [16] show that a hardware implementation retains the self-adaptation capabilities. Their implementation has a period of 7.6 ns.

For the reasons given in the preceding paragraphs, we propose to use the XCS as the evaluator of a generic self-adaptation method. The method covers both design time and run time. At design time, the XCS learns a preferably optimal set of rules for the problem at hand using a software simulation of the SoC. At run time, the learned rules will be loaded into a memory, on which the hardware implementation of the XCS will operate, choosing the actions it learned at design time and self-adapting to the actual manifestation of the chip (compensating for the differences between the simulation model and the actual realization of the chip). The method combines the advantages of both software and hardware implementations: at design time, where a lot of resources are available, the system learns fast and efficiently, while at run time, the system reacts quickly, transparently and independent of the applications that

run on the SoC. After a short introduction to the XCS, we will present the experimental setup and the results of our analysis in the following sections.

For details beyond the following short introduction to the XCS, we refer to the literature [11], [17], [18]. The XCS learns a set of optimal classifiers (or rules), each of which consists of a condition, an action, a reward prediction and some other house keeping values, most notably the accuracy of the reward prediction. The condition and the action are bit strings, where the condition additionally uses a wild card symbol to cover several cases at once. At each learning step, the XCS matches the conditions in the classifier set with the monitor signal and notes the actions and reward predictions that each classifier proposes. It then chooses the action which promises the highest reward, and, after the action has been applied by the actuator, receives the reward for the action. Based on this feedback, the XCS adjusts its reward predictions and classifier set, which, in the ideal case, leads to a classifier set that results in a maximum reward for any monitored situation and, at the same time, is as small as possible [14]. The XCS uses a genetic algorithm based on the reward prediction accuracies to generate new, possibly better suited classifiers, a feature which distinguishes it from other learning classifier systems and which is supposed to be the reason for its generally good performance.

3. Experimental Setup

This section describes the experimental setup that we use to analyze our proposed generic self-adaptation method.

As we are neither aware of benchmarks for chip-level self-adaptation, nor could we find any, we define a core-allocation problem to evaluate the basic performance, the self-adaptation capabilities and the costs of our proposed generic self-adaptation system. The problem is motivated by the advent of many-core systems [19], in which it is favorable to allocate several cores for the same process to overcome reliability issues with single cores and where the chip applications can profit instantly from a hardware-based solution without the need to adjust software. The problem definition is guided by the following reasoning. To determine the basic performance, the problem is simple enough so that we know the optimal solution, yet difficult enough so that a random solution is most likely invalid. Furthermore, the problem's degree of difficulty can be increased easily and steadily, although we stick to the basic version for this first evaluation of a generic self-adaptation system. To determine the self-adaptation capabilities, the core-allocation problem allows the simulation of unforeseen events. Lastly, it also allows to evaluate the generalization capabilities of the proposed system, a capability that is helpful to reduce design efforts.

We define the *n-out-of-c cores allocation problem* as follows: given c cores, select n ($n \leq c$) cores, which are to be allocated for a process, by indicating one of the $\binom{c}{n}$ possible allocations, while some of the c cores are known to be already occupied. The input interface of the self-adaptation system (coming from the monitors) consists of c signals that indicate the free and occupied cores. The output interface of the self-adaptation system (going to the actuator) consists of $\lceil \log_2 \left(\binom{c}{n} + 1 \right) \rceil$ signals that indicate either one of the $\binom{c}{n}$ possible allocations or that there is no valid allocation (e.g., when all cores are occupied). We say an allocation is *valid* or *correct*, if it allocates only free cores. The problem can easily be made more difficult by imposing additional constraints on the cores that are to be allocated; for example, the cores can be required to have a maximum distance to already occupied cores to avoid hot spots, they can be required to minimize communication latencies of the process, or even a combination thereof. As already mentioned, though, we stick to the basic version for the first evaluation of a generic self-adaptation system and note that additional constraints can be easily added by altering the reward function, which the designer is supposed to provide.

The first set of experiments represents the design-time learning of the evaluator and shows the basic performance level that the generic self-adaptation system can reach. The next set of experiments represents the run-time situation with unforeseen events that require self-adaptation of the evaluator. The last set of experiments represents the design-time and run-time learning of the evaluator during a restricted design process.

The experiments run as follows. During each learning step, a random number r of the c cores gets occupied ($0 \leq r \leq c$). This occupation is then presented to the evaluator as a monitor input. Based on this monitor input, the evaluator chooses an action, which is an index into a given list of all possible allocations. For example, for the three-out-of-ten allocation problem, there exist $2^{10} = 1024$ different core occupations, each of which has the same probability to be presented to the evaluator. The evaluator in turn emits an action that either points to one of the $\binom{10}{3} = 120$ possible allocations or is a special action indicating that there is no valid allocation. The evaluator receives a reward if its chosen action is valid, and no reward if its chosen action is invalid.

We will quantify the *correctness rate* and estimate a relative measure of the *chip area* of the evaluator. The correctness rate describes how many of the allocations that the evaluator proposes are actually correct (valid). A perfect evaluator will have a correctness rate of 100%. We will use the correctness rate to determine the performance of the evaluator in the basic problem and how well it can self-adapt. We use the number of bits that are necessary to store the evaluator's knowledge as a relative measure for the chip area as it usually dominates the logic part of the evaluator.

In order to judge the results of our generic self-adaptation system, we need other systems for comparison. However, as our purpose is to investigate a *generic* self-adaptation method, we cannot consider systems that have been specially designed for the core-allocation problem, as this would counteract our intentions. Therefore, we consider the following two evaluators for comparison: *random allocation* and *lookup table*. Both evaluators fit the generic interface presented in Figure 1

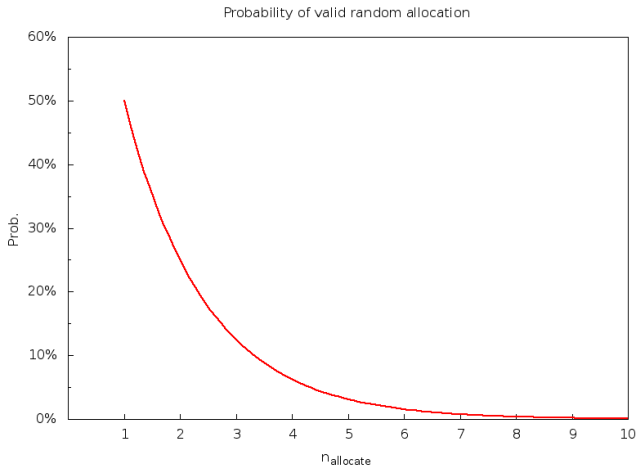

Fig. 2. Probability that a random allocation is valid.

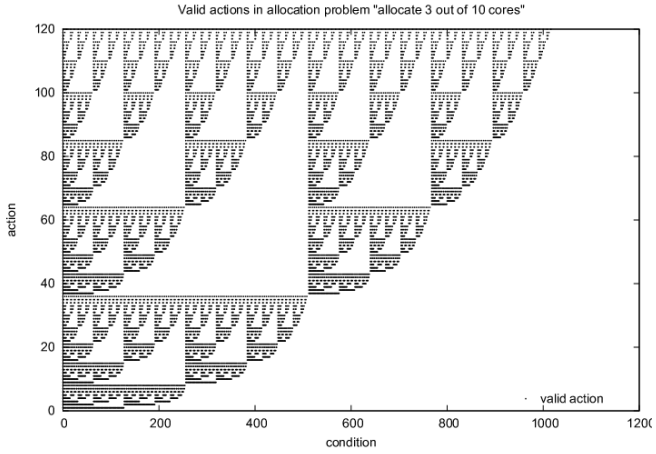

Fig. 3. Reward function for the three-out-of-ten cores allocation problem.

and are ignorant of the meaning of the monitor inputs. The random-allocation evaluator randomly chooses an allocation, independent of the actual monitor input. It represents the lower bound of the correction rate of the basic case. The lookup table (LUT) represents a perfect system as often encountered in worst-case designs with no self-adaptation capabilities: a memory that is addressed by the monitor input, at which position it contains one of the optimal actions that the designer has chosen just for this monitor input. The LUT represents the lower bound of the correction rate during self-adaptation and serves as a comparison for the chip area. Although neither the random-allocation evaluator nor the LUT are realistic candidates for a generic self-adaptation system, they fulfill the requirement that they are not specially designed for our evaluation problem.

From a generic self-adaptation point-of-view, the core allocation problem is not trivial. As can be seen in Figure 2, the self-adaptation mechanism must perform considerably better than random allocation: the probability that the random-allocation evaluator chooses a valid action drops exponentially with n. This probability is independent of the total number of cores c. The probability that a random allocation is valid lies at about 12% for allocating three cores and only at about 3% for allocating five cores.

Furthermore, the valid allocations are distributed unevenly, as depicted in Figure 3, which shows the reward function for the three-out-of-ten cores allocation problem. The x-axis shows the $2^{10} = 1024$ possible occupations of the ten cores. The y-axis shows the $\binom{10}{3} = 120$ possible core allocations as actions. The diagram contains a dot for every valid action under the given occupation condition[1]. For extended versions of the problem, the reward could scale with the goodness of the action, for example with the distance to hot spots.

1. The dots are enlarged for visibility; actually, they cover about 12% of the chart area, corresponding to the previously mentioned probability that a random allocation of three cores is valid.

4. Results

We first evaluate the basic performance of the XCS with the basic core allocation problem as described in the preceding section. We then test its self-adaptation capabilities by letting cores fail, that is, although the failed core is monitored as free, the XCS cannot allocate it. Finally, we let the XCS learn based on only 25% and 5% of all possible core occupations to reduce the time needed to complete the design of the generic self-adaptation system. All experiments are performed as software simulations.

Repeated simulations show similar patterns like we present in the following subsections and lead to the same conclusions. For each simulation run of the XCS, the results show the average values of the last 500 learning steps based on exploitation (i.e., the actions are based on acquired knowledge). We use the C implementation of the XCS as reported in [20], version 1.2. All experiments were run with the default values with the following changes (see [18] for a definition): $N = 10\,000$, $\alpha = 1$, $\gamma = 0.8$, $\gamma_{\text{fitness}} = 0.5$, $\theta_{\text{GA}} = 250$, $\xi_{\text{GA}} = 0.1$, $\mu_{\text{GA}} = 0.1$, $P_{\#} = 0.4$, $p_{\text{explr}} = 0.2$, with activated generalization mutation, Moyenne Adaptive Modifiée (MAM), GA subsumption and action-set subsumption. We chose the parameters to balance keeping acquired knowledge and discovering new classifiers. Each complete set of simulation experiments took about one hour on contemporary hardware, parallelized over eight CPU cores.

4.1. Basic core allocation problem

The first experiments represent the design-time learning of the XCS, in preparation of the run-time system. In this experiment, the XCS shall learn to allocate free cores in the basic core-allocation problem. We evaluate the correctness rate that the XCS reaches to determine its basic performance.

The results for the three- and five-out-of-ten cores allocation problems are shown in Figure 4. The charts show the number

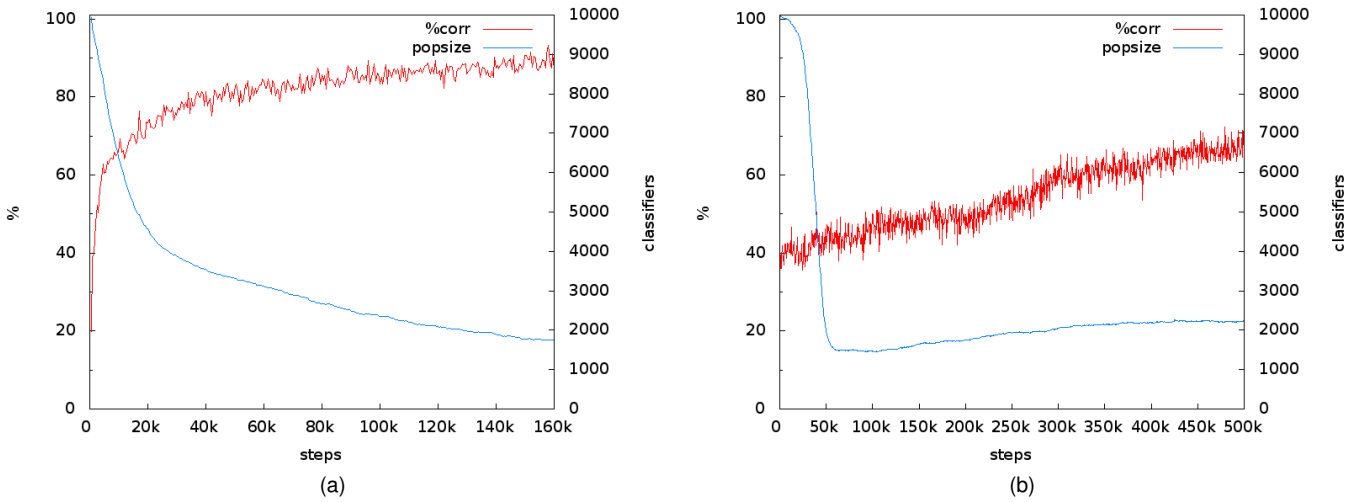

Fig. 4. Design-time learning of the XCS. (a) Solving the three-out-of-ten allocation problem. (b) Solving the five-out-of-ten allocation problem.

of learning steps at the x-axis, the correctness rate on the left y-axis (labeled '%') and the number of classifiers on the right y-axis (labeled 'classifiers'). Figure 4a shows that, initially, the correctness rate is very low, as at this stage the XCS has not acquired any knowledge and thus randomly tries various actions. Alternatively, the designer could have provided his expert knowledge for an increased initial correctness rate, in which case the XCS would start its learning process from there. The correctness rate increases rapidly until it reaches a level of about 90%, which is significantly higher than the correctness rate of random allocation of 12% (see Figure 2). We observe a similar pattern for the five-out-of-ten allocation problem, shown in Figure 4b. However, the slope of the correctness rate is smaller and the XCS reaches a correctness rate of only about 70%, which is significantly higher than the correctness rate of random allocation of 3%, but which leaves room for improvement. The correctness rate of the LUT for this experiment is 100% by design (not depicted).

The population size initially increases as the XCS randomly creates classifiers to cover large parts of the problem space. Afterwards, the XCS combines similar classifiers using the wild card symbol. The population size is about 1 800 for the three-out-of-ten allocation problem and about 2 100 for the five-out-of-ten allocation problem when the correctness rate does not improve further.

We run the experiments for all combinations of c and n_{allocate} with $1 \leq n_{\text{allocate}} \leq c \leq 10$. The contour plot in Figure 5 depicts the correctness rate of the XCS at the end of the learning process. We ended the learning process when the correctness rate did not improve further, which happened after 500 000 learning steps or earlier. The x-axis shows the number of available cores (labeled 'cores') and the y-axis shows the number of cores to be allocated (labeled 'n_{allocate}'). The contour plot contains an isoline for every five percentage points.

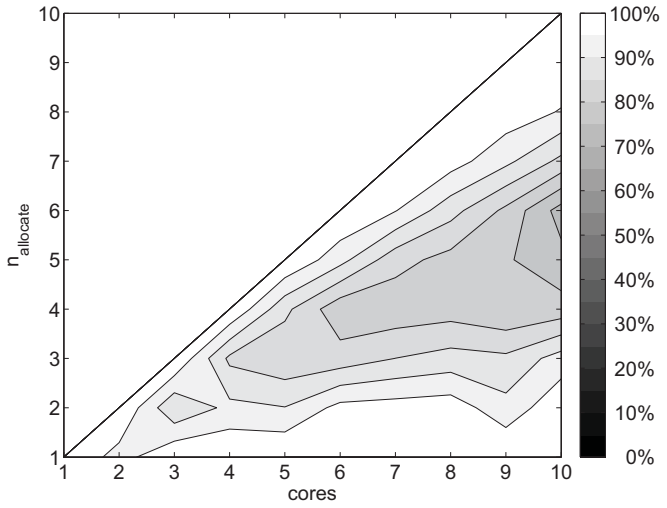

Fig. 5. Probability of a valid allocation by the XCS in the basic core allocation problem.

The XCS reaches a correctness rate of 90% or more for the simple problem configurations where the XCS has to allocate either only a few cores (white area near the x-axis) or almost all cores (white area below the main diagonal). In between, the number of possible allocations is large, which means that the number of possible actions is large and the XCS has a harder time to figure out valid actions. However, the correctness rate reaches 68% (for the six-out-of-ten problem) or more. Comparing with Figure 2, we find that the correctness rate of the XCS is considerably higher than the correctness rate of the random-allocation evaluator, which reaches a correctness rate of only 50% or less. For example, for the five-out-of-ten allocation problem, the XCS has a correctness rate of 71% whereas the random-allocation evaluator has a correctness rate of only 3%.

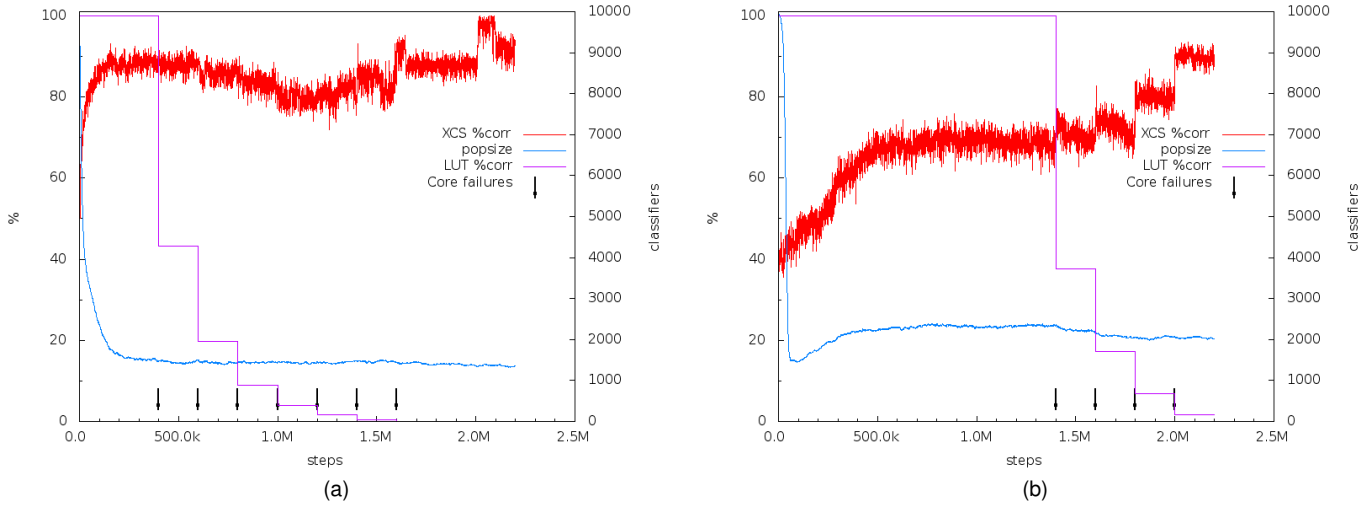

Fig. 6. Self-adaptation to failing cores of the XCS at run time. (a) Three-out-of-ten allocation problem. (b) Five-out-of-ten allocation problem.

4.2. Self-adaptation to failing cores

We next analyze the self-adaptation capabilities of XCS at run-time. We assume that the basic performance at run-time is the same as presented in Section 4.1.

Of course, no evaluator can self-adapt to every unforeseen event. However, it is desirable that the evaluator has the capability to tolerate at least some unforeseen events, so that it helps to alleviate the design process. In the following experiment, we simulate the run-time behavior of the XCS and analyze whether the XCS can tolerate the total failure of one ore more cores, although the XCS has not had the opportunity to learn actions for these events at design time. A failed core is monitored as free, but an allocation of this core fails, that is, although the core looks available to the XCS, it cannot be allocated. This would be a typical symptom if the engineer did not consider the failure of a core at design time and did not instruct the monitors to report failed cores as not available. We assume that a failed core never recovers.

Figure 6 shows the correctness rates of the XCS and the LUT for the three- and five-out-of-ten allocation problem, respectively, during core failure. The arrows mark when a random core failed. For the XCS, the result of a single run and a sequence of core failures is depicted; other runs with different sequences of core failure show similar patterns. For the LUT, the average over all possible LUTs (each of which uses a different optimal action for a given monitor input) is depicted, which is an analytical result. In Figure 6a, the first core is instructed to fail after 400 000 learning steps (when the correctness rate does not improve further, which represents the situation after design-time learning) and after that, every 200 000 steps another core fails until three available cores are left over (at 1.7 million learning steps)[2]. If more cores failed, the system would become unusable in the intended sense. From the figure we note that the correctness rate for the XCS drops from about 90% to 80% when half of the ten cores have total failures. After that, the correctness rate increases again to the usual level of 90%. In Figure 6b, the first core is instructed to fail after 1.4 million learning steps and after that, every 200 000 steps another core failed until five cores are left over (at 1.6 million learning steps). From the figure we note that the correctness rate initially stays at the usual 70% level and increases when only six functioning cores are left over. The correctness rate for the LUT drops monotonically in both cases, as core failures have not been considered during its design.

We run the experiments for all combinations of c and n_{allocate} with $1 \leq n_{\text{allocate}} \leq c \leq 10$ and note the correctness rate at 100 000 steps after the first core has failed. Figure 7a depicts the resulting correctness rate for the XCS. From the figure we note that the correctness rate of the XCS after one core has failed is at about 90% when either a few cores (near the x-axis) or almost all cores (below the main diagonal) have to be allocated. In between, the correctness rate reaches almost 70% (for the five-out-of-ten problem) or more. Comparing with Figure 5, we find that the XCS is able to maintain the correctness rate of the basic core allocation problem and still stays above the correctness rate of the random-allocation evaluator (not depicted).

In contrast, the LUT is not able to maintain its correctness rate, as shown in Figure 7b, which is an analytical result. The correctness rate of the LUT decreases significantly after one core has failed. While for the basic core allocation problem the correctness rate of the LUT is 100% for all combinations by design, the correctness rate after one core has failed drops to as low as 25%, as core failures had not been considered explicitly during the design of the LUT. For the three-out-of-ten and five-out-of-ten allocation problems, the correctness rates drop to 43% and 38%, respectively.

2. The number of learning steps between core failures does not have a significant effect on the results; we choose it to be large enough so that the system can encounter all possible situations and we thus get meaningful correctness rates between core failures.

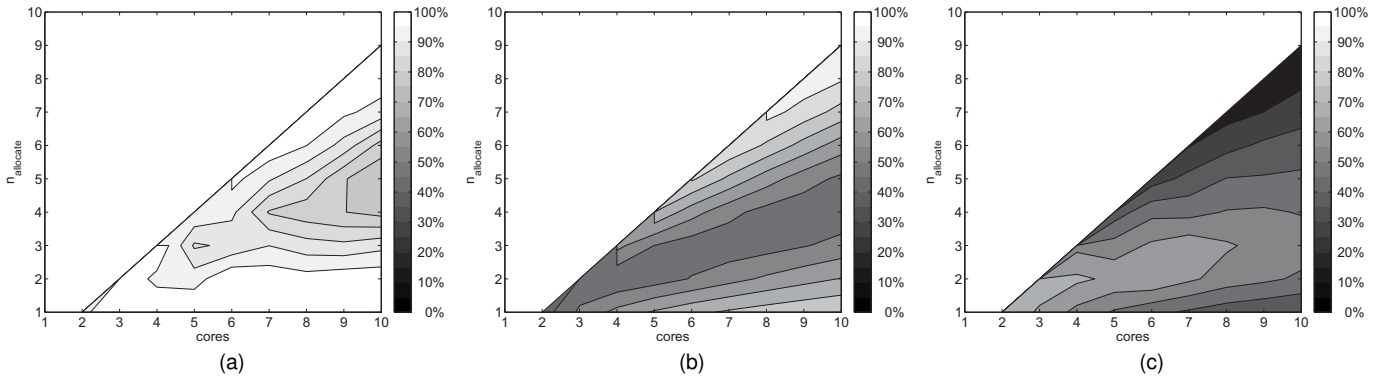

Fig. 7. Probability of a valid allocation after one core has failed. (a) XCS. (b) LUT. (c) Difference between XCS and LUT in percentage points.

Figure 7c illustrates the difference between the correctness rates of the XCS and the LUT in percentage points for easier comparison. We note that the correctness rate of the XCS is always better than the correctness rate of the LUT. For the three-out-of-ten and five-out-of-ten allocation problems, the correctness rates of the XCS are 45 and 32 percentage points larger than the correctness rate of the LUT, respectively.

We conclude that the XCS is able to tolerate core failures, although this has not been considered during its design, and that the XCS can quickly self-adapt to the total failure of a considerably large number of cores. The correctness rate does not drop significantly and usually stays almost constant.

4.3. Generalization after restricted learning

In order to find the optimal action for every possible situation, the designer has to reason about it and/or simulate it. For the core-allocation problem, the simulations run quickly, but we can easily imagine simulations that take longer, for example when simulating the communication of an application. If a single simulation takes long and the configuration for one simulation depends on the outcome of another simulation, it becomes prohibitive to run simulations for all possible states of the chip. Thus, it is desirable to keep the number of necessary design-time simulations at a minimum.

The last set of experiments covers both the design time and the run time of the generic self-adaptation method. At design time, we present the evaluator only a randomly chosen, but fixed subset of all possible core occupations. At run time, of course all core allocations are possible. We analyze whether the XCS can generalize from the restricted design-time set to the complete run-time set.

The results are shown in Figure 8. The first 500 000 learning steps represent the design-time learning, where the XCS sees only a randomly chosen but fixed subset of 25% of all possible core occupations. The following learning steps represent the run-time situation, where all core occupations occur. Figure 8a and Figure 8b show the results for the three- and five-out-of-ten allocation problem, respectively. We note from both figures, that during design-time learning, the correctness rates reach the usual 90% and 70% level. After switching to run-time learning, the correctness rate drops slightly to about 85% and 65%, respectively, and stays there. We also note that the variation of the correctness rate in Figure 9b (the 'width' of the correctness rate line) is larger than when learning is based on all possible occupations. We simulate all other combinations of available and to be allocated cores, with similar results (not depicted). All combinations have a performance of what we have shown for the five-out-of-ten allocation problem or better.

We conclude that the XCS is able to generalize from restricted learning, offering the opportunity to learn only from a smaller subset at design time and let the XCS generalize and self-adapt to the full set at run time. As the simulation of the smaller subset does not take as long as the simulation of all possible occupations, this capability offers the possibility to keep a shorter time to market.

We restrict learning further to find a lower bound for the amount of occupations needed during design-time learning. Figure 9a and Figure 9b show the result for the three- and five-out-of-ten allocation problem, respectively. This time, the XCS sees only a random but fixed sample of 5% of all possible occupations. From the figures we note that the variation in the correctness rate increases considerably. For the three-out-of-ten allocation problem, the variation of the correctness rate covers about 10%, while for the five-out-of-ten problem, the variation of the correctness rate covers 20% on average and 40% between the peaks. Clearly, learning based on only 5% of all possible occupations pushes the XCS to its limits. Yet, at run time, when the XCS sees all possible occupations, the variation of the correctness rate decreases considerably and the correctness rate reaches the usual levels of 90% for the three-out-of-ten allocation problem (Figure 9a) and 70% for the five-out-of-ten allocation problem (Figure 9b).

4.4. Costs

A generic self-adaptation methodology involves costs during design time and additional chip area, of course. Quantifying

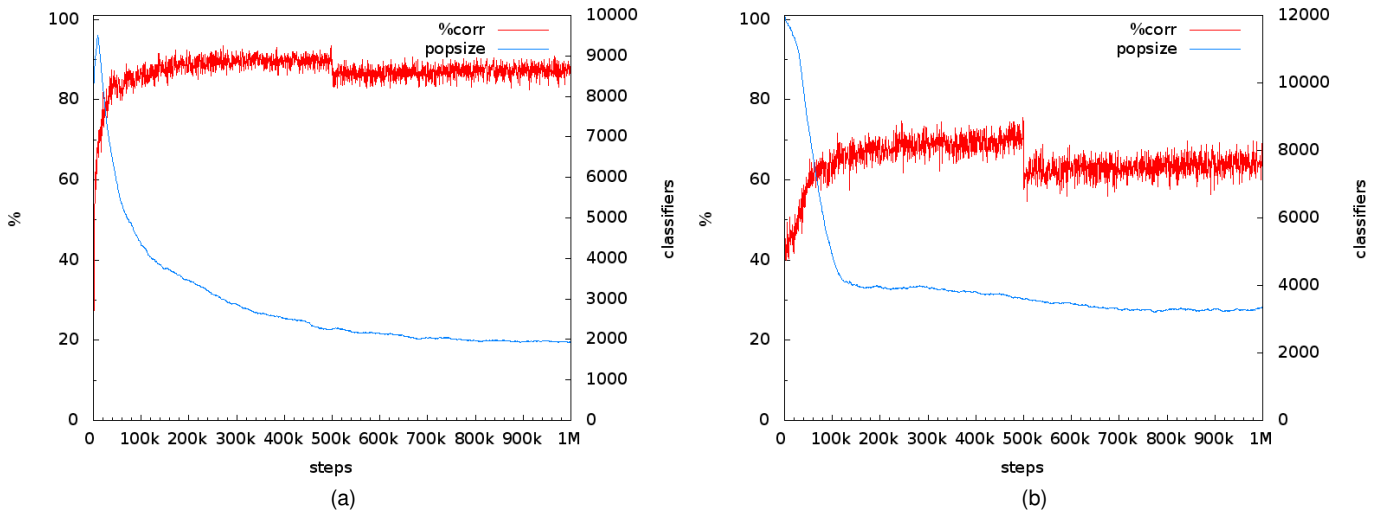

Fig. 8. Run-time generalization from 25%-restricted design-time learning. (a) Three-out-of-ten allocation problem. (b) Five-out-of-ten allocation problem.

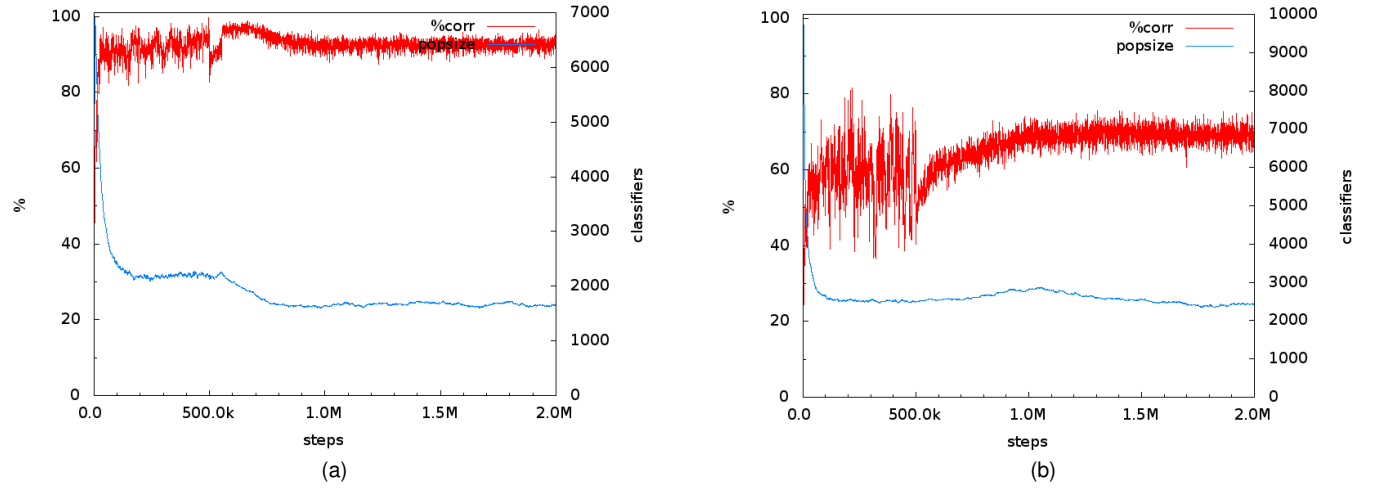

Fig. 9. Run-time generalization from 5%-restricted design-time learning. (a) Three-out-of-ten allocation problem. (b) Five-out-of-ten allocation problem.

the costs involved during design time is difficult, though. In any case, we can qualitatively argue why we think the design time costs are less compared to designing a specific solution: the only parts of the XCS that change between different designs are the input and output vectors and the reward function. For the input and output vectors, the designer has to decide what monitor signals to use, their discretization, and what actions to perform. These are the same tasks as for any problem-solving design and the designer usually knows the solutions.

Different monitor signals and actions will result in different bit lengths of the input and output vectors, and the design has to be changed accordingly. This is a straightforward task, as it involves only regular structures [16]. Furthermore, designs that use the same number of input and output bits can be reused for different designs, as the XCS is ignorant to the meaning of the individual bits. For example, if two designs use an eight bit input vector to generate a four bit output vector, the same XCS design can be used (with different classifier tables, of course). Moreover, as the XCS can learn the significant bits in the input vector, we think it is possible that a particular XCS design can be used for a problem that uses less input and output bits (e.g., an XCS design with an eight bit input and four bit output vector can be used for a problem with only seven bit input and three bit output). Although we have not conducted any experiments for this yet, the resulting classifier tables of the experiments that we presented here and [14], which reports that the XCS evolves complete and minimal rule representations, gives us reason for this expectation. For the actual table sizes, see the following.

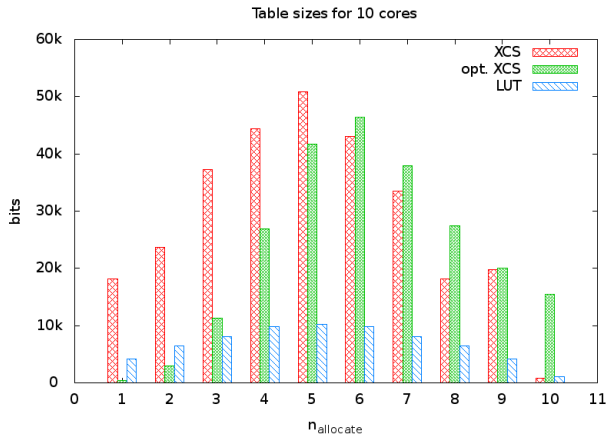

Fig. 10. Table sizes for the XCS, the optimal XCS and the LUT.

The largest part of the design effort for the XCS is designing the circuitry for the reward function. The reward function will read in the current monitor signals and perform the calculations necessary to get the assessment of the achieved state of the chip. Although the complexity of this tasks depends on the reward function, the design of calculating circuits is well-understood and should not pose a great challenge to a designer. In particular, we expect the design of a calculating circuit to be less difficult than the design of a special-purpose circuit to solve the problem at hand.

The necessary chip area to implement the XCS is mainly determined by the number of classifiers that the XCS generated to obtain the correctness rate reported previously [16]. From the experiments on the basic core allocation problem (see Section 4.1), we collect the population size after the correctness rate did not improve any further. Figure 10 shows the resulting population sizes for the problems with a total of $c = 10$ cores. The x-axis identifies the number n_{allocate} of cores to be allocated and the y-axis shows the number of bits that are necessary to store the resulting classifier population. The figure shows the table sizes for the actual XCS (red), a theoretically optimal XCS (opt. XCS, green), and a fixed lookup table (LUT, blue). For the actual XCS, each classifier consists of $c = 10$ bits for the condition, $\lceil \log_2 \left(\binom{c}{n} + 1 \right) \rceil$ bits for the action[3] and four bits for the predicted reward [16]. The optimal XCS contains the minimum amount of classifiers to guarantee a correctness rate of 100% and shows the lower bound for the size of a classifier table of a perfect XCS. The LUT is supposed to be engineered by a designer. It is a memory that is indexed by the current monitor state and contains the optimal action for each monitor state[4].

Figure 10 shows that the XCS needs about five times as many bits than a lookup table. For small numbers of n_{allocate}, the actual XCS is also significantly larger than the optimal XCS, while for large numbers, the table sizes of the actual and the optimal XCS are about the same. In any case, the data for the optimal XCS shows that the XCS will be considerably larger than a lookup table.

The main reason for the large difference between the table size of the actual XCS and the table size of the LUT is that the XCS learns all possible and impossible (invalid) actions for a given condition, while the LUT contains only one valid action for a given condition. The XCS uses the extra information to quickly adapt to unforeseen events (based on the alternative actions) or generalize from the learned classifier set (based on the invalid actions).

3. We need one more action than possible combinations for the special action when there is no valid allocation.

4. Or, more precisely, it contains the index of the optimal action.

5. Conclusion

The increasing integration density and design complexity of nano-scale chips require methods and systems to mitigate the design effort and increase design reuse. While special-purpose self-adaptation systems help, their special-purpose circuit put an extra burden on the design effort and their design reuse rate is low. We argue that a generic self-adaptation method can remove this burden and help mitigate the design efforts.

We showed that the basic performance of a generic self-adaptation methodology that uses the XCS as an evaluator is only 10% less than the performance of a perfect system in the average case, and only 32% less in the worst case. In any case, the performance is considerably better than random allocation. We further showed the self-adaptation capabilities of an XCS-based generic self-adaptation system. Even when half of the initially available cores have total failures, the system keeps an almost constantly high correctness rate, while the correctness rate of an initially perfect but non-adaptive system decreases considerably. Additionally, we showed that our proposed generic self-adaptation system is able to generalize from restricted learning, which allows the possibility to shorten the time needed to complete a design. Presenting and simulating only 25% of all possible run-time situations at design time showed to be enough to let the run-time system self-adapt and reach correctness rates that are similar to the correctness rates when all possible run-time situations are presented at design time. Finally, we discussed that the learning capabilities and the generality of a generic self-adaptation method helps to reduce design efforts.

In summary, the results show that our proposed generic self-adaptation methodology provides a viable alternative to special-purpose solutions, as the designer can let the XCS learn the optimal actions, the run-time system is robust against unforeseen events and the time needed to complete a design can be reduced.

The benefits come with the cost of designing the reward function and of increased chip area, up to five times the size of an optimal lookup table. However, increasing integration density and shorter time to markets may lead the designer to decide to use a generic self-adaptation methodology that generates its solutions automatically. This is a middle ground between the poor chip performance of worst case

designs of nano-scale chips and a prolonged design process in better-than-worst-case designs incorporating manually designed special-purpose solutions.

Acknowledgments

We thank the team of Prof. Herkersdorf (TU Munich) for fruitful discussions. We also thank Thomas Schweizer (University of Tübingen) for his valuable comments on an early draft of this paper. Furthermore, we thank François-Xavier Morel, École Polytechnique, Paris, for providing the results on the correctness rate of the LUT. A.B. thanks Eva Gottwald for fruitful discussions and moral support.

References

[1] International Roadmap Committee, "International Technology Roadmap for Semiconductors," http://www.itrs.net/reports.html, 2008.

[2] K. Bernstein, D. Frank, A. Gattiker, W. Haensch, B. Ji, S. Nassif, E. Nowak, D. Pearson, and N. Rohrer, "High-performance CMOS variability in the 65-nm regime and beyond," *IBM Journal of Research and Development*, vol. 50, no. 4/5, p. 433, 2006.

[3] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 338–342.

[4] A. Agarwal, V. Zolotov, and D. Blaauw, "Statistical clock skew analysis considering intra-die process variations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1231 –1242, 2004.

[5] C. Schlunder, R. Brederlow, B. Ankele, A. Lill, K. Goser, R. Thewes, I. Technol, and G. Munich, "On the degradation of p-MOSFETs in analog and RF circuits under inhomogeneous negative bias temperature stress," in *41st Annual IEEE International Reliability Physics Symposium Proceedings*, 2003, pp. 5–10.

[6] V. Narayanan and Y. Xie, "Reliability Concerns in Embedded System Designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006. [Online]. Available: http://www.cse.psu.edu/~yuanxie/Papers/ComputerMag2006.pdf

[7] S. Borkar, "Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, November/December 2005.

[8] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2005, pp. 2–7.

[9] A. Tiwari, S. R. Sarangi, and J. Torrellas, "ReCycle: pipeline adaptation to tolerate process variation," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2007, pp. 323–334.

[10] E. Karl, D. Sylvester, and D. Blaauw, "Timing error correction techniques for voltage-scalable on-chip memories," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, vol. 4, 2005, pp. 3563 –3566.

[11] S. W. Wilson, "Generalization in the XCS Classifier System," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 22-25 1998, pp. 665–674. [Online]. Available: http://citeseer.ist.psu.edu/wilson98generalization.html

[12] R. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[13] J. H. Holland, "Adaptation," in *Progress in theoretical biology*, R. Rosen and F. M. Snell, Eds. New York: Academic Press, 1976, pp. 263–293.

[14] T. Kovacs, "XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions." in *Soft Computing in Engineering Design and Manufacturing*, Roy, Chawdhry, and Pant, Eds. Springer-Verlag, London, 1997, pp. 59–68. [Online]. Available: http://citeseer.ist.psu.edu/article/kovacs97xcs.html

[15] C. Bolchini, P. Ferrandi, P. L. Lanzi, and F. Salice, "Evolving classifiers on field programmable gate arrays: Migrating XCS to FPGAs," *Journal of Systems Architecture*, vol. 52, no. 8-9, pp. 516–533, 2006.

[16] J. Zeppenfeld, A. Bouajila, W. Stechele, and A. Herkersdorf, "Learning Classifier Tables for Autonomic Systems on Chip." in *GI Jahrestagung (2)*, ser. LNI, H.-G. Hegering, A. Lehmann, H. J. Ohlbach, and C. Scheideler, Eds., vol. 134. GI, 2008, pp. 771–778.

[17] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995. [Online]. Available: citeseer.ist.psu.edu/wilson95classifier.html

[18] M. Butz and S. W. Wilson, "An Algorithmic Description of XCS," in *IWLCS '00: Revised Papers from the Third International Workshop on Advances in Learning Classifier Systems*, ser. Lecture Notes in Artificial Intelligence, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds., no. 2321. London, UK: Springer-Verlag, 2001, pp. 253–272. [Online]. Available: http://citeseer.ist.psu.edu/313131.html

[19] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual conference on Design automation*. ACM New York, NY, USA, 2007, pp. 746–749.

[20] M. V. Butz, "An Implementation of the XCS classifier system in C," The Illinois Genetic Algorithms Laboratory, Tech. Rep. 99021, 1999.