Combining Software and Hardware LCS for Lightweight On-Chip Learning

Andreas Bernauer[1], Johannes Zeppenfeld[2], Oliver Bringmann[3], Andreas Herkersdorf[2], and Wolfgang Rosenstiel[1]

[1] University of Tübingen, 72076 Tübingen, Germany
bernauer@informatik.uni-tuebingen.de
[2] Technische Universität München, 80290 München, Germany
[3] Forschungszentrum Informatik, 76131 Karlsruhe, Germany

Abstract. In this paper we present a novel two-stage method to realize a lightweight but very capable hardware implementation of a Learning Classifier System for on-chip learning. Learning Classifier Systems (LCS) allow taking good run-time decisions, but current hardware implementations are either large or have limited learning capabilities.
In this work, we combine the capabilities of a software-based LCS, the XCS, with a lightweight hardware implementation, the LCT, retaining the benefits of both. We compare our method with other LCS implementations using the multiplexer problem and evaluate it with two chip-related problems, run-time task allocation and SoC component parameterization. In all three problem sets, we find that the learning and self-adaptation capabilities are comparable to a full-fledged system, but with the added benefits of a lightweight hardware implementation, namely small area size and quick response time. Given our work, autonomous chips based on Learning Classifier Systems become feasible.

Keywords: System-on-Chip, Learning Classifier System, XCS

1 Introduction

As the number of functions integrated in a single chip increases, the complexity of a chip grows significantly. Furthermore, increasing transistor variability [4, 6], process variation [1], and degradation effects [18] make it increasingly difficult to ensure the reliability of the chip [16]. The International Technology Roadmap for Semiconductors (ITRS) [13] estimates that until 2015, up to 70% of a chip's design must be reused to keep up with the increasing complexity.

Autonomic System-on-Chip (ASoC) [15] add a logical, autonomic layer to contemporary SoCs that helps the designer to manage the complexity and reliability issues: decisions that are hard to take at design time because many parameters are uncertain, can be taken at run time by the autonomic layer. Learning Classifier Systems (LCS) have been shown to be able to take the right run-time decisions [3, 2] and even adapt to events that due to the chip complexity have not been foreseen at design time. LCS use a genetic algorithm and

reinforcement learning to evolve a set of rules, the interaction of which propose a preferably optimal action to any situation the chip may encounter. Although LCS allow very capable systems for autonomous run-time decisions and self-adaptation, current hardware implementations either require large portions of the chip [5], increasing total chip costs, or have limited learning capabilities [24].

In this paper, we present a novel two-stage method to realize an on-chip Learning Classifier System (LCS) that is small, takes the good run-time decisions, and can adapt to unexpected events. In the first stage at design time, we learn a rule set in software using a particular LCS, the XCS [23]. In the second stage, we use the rule set to initialize the lightweight LCS hardware implementation LCT [24]. The idea is that the XCS learns just enough rules so that the LCT can adapt to the actual manifestation and conditions of a particular chip and even to unexpected events, albeit in a limited way.

We first compare our method to other LCS implementations using the multiplexer problem, a traditional testbed for LCS [23], and then apply it to two chip-related problems, namely task-allocation and SoC component parameterization. We show that the LCT can adequately learn and still react to unexpected events. To the best of our knowledge, this is the first study of a lightweight but still capable hardware implementation of an LCS. We think that our work makes using LCS to control chips conceivable.

This work is structured as follows. Section 2 gives an overview of related work. Section 3 introduces the XCS and the hardware implementation LCT. Section 4 describes our proposed method. Section 5 presents the three benchmarks multiplexer, task-allocation and SoC component parameterization that we use to assess our method. Section 6 shows the results of our assessment and Section 7 concludes this paper.

2 Related Work

Learning Classifier Systems were originally introduced in [12]. The XCS was first presented in [21] and later refined in [23]. The XCS has been used in a large range of learning and classification problems, including controlling a robotic mouse [10], a system-on-chip (SoC) [3], the lights of a traffic junction [17], and for finding suitable partitions in hardware-software codesign [11]. A first hardware implementation of an XCS has been presented in [5], named XCS_i, which uses fixed-point arithmetic. The implementation shows good learning rates of the XCS_i, but is quite large. In [24], the authors present an optimized hardware implementation of an LCS, called the Learning Classifier Table (LCT), which is small but has no mechanism to create new classifiers. Using a hand-crafted initial rule set, the authors show that the LCT can adjust the frequency of a SoC according to a given objective function.

The most popular machine learning algorithms for which hardware implementations exist are neural networks [19, 9] and, more recently, support vector machines [14]. Along with the fact that for these systems, "the actual rules im-

plemented [are] not apparent" [19], their implementations are about five times as large as the LCT [14].

3 XCS and LCT

We briefly describe the XCS and LCT and refer to [22, 23, 7, 24] for further details.

The XCS learns a minimal set of *classifiers* (or *rules*) the interaction of which, in the ideal case, provide an optimal response (called *action*) for a given situation. The learning is based on a genetic algorithm and reinforcement learning. Each classifier (or rule) consists of a condition, an action, a reward prediction, the reward prediction accuracy, and some other house keeping values. The condition is a string of bits ('0', '1', and the don't-care symbol '#'). At each learning step, the XCS matches the input signal with the condition of each classifier and notes the actions and accuracy-weighted reward predictions that each classifier proposes. The XCS then selects an action to apply: in the exploit mode, it chooses the action that promises the highest reward, while in the explore mode, it chooses a random action to find new alternatives. After the action has been applied, the XCS receives a reward depending on the new state and updates its reward predictions and classifier set accordingly. After some number of iterations, the genetic algorithm repeatedly creates new, possibly better suited rules.

The LCT consists of a memory, which holds a fixed number of classifiers, and hardware-based mechanisms for action lookup and fitness update. There is no mechanism to generate new classifiers. The classifiers in the LCT consist only of a condition, an action and a fitness, similar to the fitness in the strength-based ZCS [20]. To realize the don't-care bits, the LCT first logically ANDs the monitor signal with a mask before comparing it with the bit value. The LCT selects the action of a matching classifier randomly according to the classifier's relative fitness (roulette-wheel selection) using weighted reservoir sampling to ensure a fixed lookup time. After receiving the reward for a previously applied action, the LCT distributes the reward r to the classifiers of the action set and updates the fitness f according to $f \leftarrow \beta r + (1 - \beta)f$ with the learning rate $0 \leq \beta \leq 1$.

4 Methodology

One major trade-off of hardware-based machine learning lies between the learning capabilities of the implementation and the allotted hardware resources: the system is either very capable but requires a lot of resources or it requires little resources but is less capable. We address this problem with the following two-stage approach:

1. At design time, the software-based XCS learns a (preferably optimal) set of rules to solve a given problem.

2. We translate the XCS rules with our `xcs2lct` too into a form that is suitable for the LCT. Initialized with these rules, the hardware-based LCT continues to learn at run time.

With this setup, we can use all the resources that are available to a capable software implementation (the XCS) and use the acquired knowledge in a lightweight hardware implementation (the LCT). The idea is that the XCS learns a rule set that allows the LCT to adapt to the actual manifestation and conditions of a particular chip and even to unexpected event, despite its limited learning capabilities.

As the chip area that is necessary to store the classifiers in memory constitutes the largest part of the LCT, we would like to minimize the number of necessary classifiers to keep the chip area requirement small. We therefore consider translating both all XCS rules to corresponding LCT rules (all-XCS translation) and only the top performing rules (top-XCS translation). The `xcs2lct` translates the rules according to the following algorithm, which ensures that the XCS and the LCT classifiers match the same input values:

```
foreach b ← xcs-rule[i] do
   if b == '#' then lct-rule[i].(mask,bit) ← ('0', '0');
   else             lct-rule[i].(mask,bit) ← ('1', b);
```

To compare our method with the base performance of the LCT, we also consider two more ways to generate LCT rules, full-constant and full-reverse. Both translations provide all possible LCT rules, that is, a complete condition-action table[4] as there is no known method to generate an appropriate rule table for the LCT. The full-constant translation initializes the rule fitness to half the maximum reward (500) and, as it is independent of the XCS rules, represents the bottom line of LCT's own learning capabilities. The full-reverse translation sets the rule fitness to the highest predicted reward of all matching XCS rules, or zero, if no XCS rule matches, and represents the combined learning capability of the LCT and the XCS.

The original action selection strategy for the LCT is roulette-wheel, which selects actions randomly according to the relative predicted reward of the matching classifiers, similar to the explore mode of the XCS. Additionally, we also consider the winner-takes-all strategy, which selects the action whose matching classifiers predict the highest reward, similar to the exploit mode of the XCS. However, unlike in the XCS, in the LCT the accuracy of the prediction does not influence the action selection.

While the XCS is usually configured to alternate between the explore and exploit mode, in our experiments the LCT uses only one of either strategies. We leave the analysis of alternating strategies in the LCT as future work.

[4] Of course, the memory requirements of the classifiers generated with full-* grow exponentially with the problem size. We use them only for comparison.

5 Experimental Setup

We use three problem types to assess our method: multiplexer [21], task allocation [2], and SoC component parameterization. Additionally, we define an unexpected event for each problem type to explore LCT's learning ability. As the XCS has already been shown to be able to solve these problem types and adapt to unexpected chip events [2], in this work we concentrate on the performance of the LCT.

The *multiplexer* problem is a typical LCS benchmark [23]. The n-multiplexer-problem is defined over binary strings of length $n = k + 2^k$. The first k bits index a bit in the remaining bits. The correct action for the LCS is the value of the indexed bit. For example, in the 6-multiplexer problem, $m_6(011101) = 0$ and $m_6(100100) = 1$. We define the *inversed multiplexer* as the unexpected event for the multiplexer, that is, the LCS is supposed to return the inversed value of the indexed bit. For example, in the inversed 6-multiplexer problem, $\overline{m}_6(011101) = 1 - m_6(011101) = 1$. We use the same XCS parameters as the full-fledged FPGA implementation of XCS presented in [5] to have comparable results: $\alpha = 0.1$, $\beta = 0.2$, $\delta = 0.1$, $\varepsilon_0 = 10$ (which is 1% of the maximum reward), $\nu = 5$, $\theta_{\mathrm{GA}} = 25$, $\chi_{\mathrm{GA}} = 0.8$, $\mu_{\mathrm{GA}} = 0.04$, $P_{\#} = 0.3$; GA subsumption is on with $\theta_{\mathrm{GAsub}} = 20$, while action set subsumption is off. We do not use generalization or niche mutation. The reported results are averages over 20 runs.

The *task-allocation* problem has been first introduced in [2] and is motivated by the advent of multi-core systems, where tasks can be run on several cores simultaneously to increase overall reliability. In the (L, i)-task-allocation problem, the LCS must allocate i available tasks on $L \geq i$ cores, some of which are known to be occupied and thus not available. The system input is a binary string of length L, where each bit represents the occupation of a particular core. There is one action for each possible allocation plus a special action that indicates that no allocation is possible (e.g., when all cores are already occupied), totaling $\binom{L}{i} + 1$ possible actions. An action is valid and returns the maximum reward if the corresponding allocation only allocates available cores; otherwise the reward is zero. The unexpected event for the task-allocation problem is the unmonitored failure of a core: although reported as available, the core cannot be occupied, and an allocation of that core returns zero reward. For the task-allocation problem, we use the XCS parameters from [2] to have comparable results, which differ from the multiplexer settings only in the following parameters: $\alpha = 1$, $\theta_{\mathrm{GA}} = 250$, $\chi_{\mathrm{GA}} = 0.1$, $\mu_{\mathrm{GA}} = 0.1$, $P_{\#} = 0.4$; GA subsumption is off. The reported results are averages over 5 runs, due to the longer simulation time for the many problem instances.

The *SoC component parameterization* problem demonstrates the ability of LCS to dynamically parameterize a system-on-chip at run time, similar to [3]. The system consists of a processing core that is subject to random load fluctuations. As the load changes, the LCS is responsible for setting the operating frequency of the core as low as possible (i.e., maintaining as high a utilization as possible), while ensuring that the core can keep up with the workload. The monitor input consists of the core's current frequency as well as its utilization.

There are five possible actions: four actions to increase or decrease the core's operating frequency by 10 or 20 MHz over a range from 50 to 200 MHz, and one action to keep the core's frequency unchanged. The reward for each action is calculated by comparing the value of a system-wide objective function before and after the action is applied. The objective function indicates how far from the designer-specified optimum of high utilization and low error rate the system is currently operating and is defined as $f_{obj} = (100\% - \text{utilization}) + \text{error\_rate}$, where a low value indicates that the system is operating near its optimum. A base reward of half the maximum reward (500) is given when the objective function returns the same value before and after the action is carried out. This is the lowest possible reward without actively worsening the system's operating state. The unexpected event for the component parameterization problem is a manufacturing defect that causes critical timing errors for operating frequencies in excess of 100 MHz. As a result, increasing the frequency above 100 MHz causes the core to cease functioning, resulting in wasted cycles for error correction and providing lower rewards to the LCS. With timing errors, the LCT must therefore learn to cap the frequency at 100 MHz, even when the workload would warrant higher operating frequencies. We use the same XCS parameters as for the task-allocation problem, except for $\alpha = 0.8$ and $P_{\#} = 0.1$. The reported results are averages over 100 runs.

We use the implementation of the XCS in the programming language C as described in [8] as the software version of XCS. We use a SystemC-based simulation model of the LCT hardware implementation described in [24], with the additional winner-takes-all strategy described in Section 4.

We compare the performance of the LCT that has been instructed using our method with the base performance of the LCT, the performance of the full-fledged hardware implementation of the XCS presented by [5], the performance of the XCS reported in [2], and the performance of the software version of the XCS. We also check whether the LCT retains the capability of LCS to adapt to unexpected events.

6 Results

In this section we present the results on the three problem types multiplexer, task-allocation, and SoC component parameterization mentioned previously.

6.1 Multiplexer

Figure 1 shows the correctness rate (x-axis) and population size (y-axis) for the 6-, 11-, 20-, and 37-multiplexer problem for all eight possible combinations of translations and action selection strategies for the LCT. Note that the x-axis starts at 70% correctness rate and that the scales of the y-axes differ. The top-XCS translation uses only classifiers that predict the maximum reward with perfect accuracy. As we aim for a small but correct LCS, in each graph lower right is better. The figures show that in the new winner-takes-all (WTA) strategy (solid

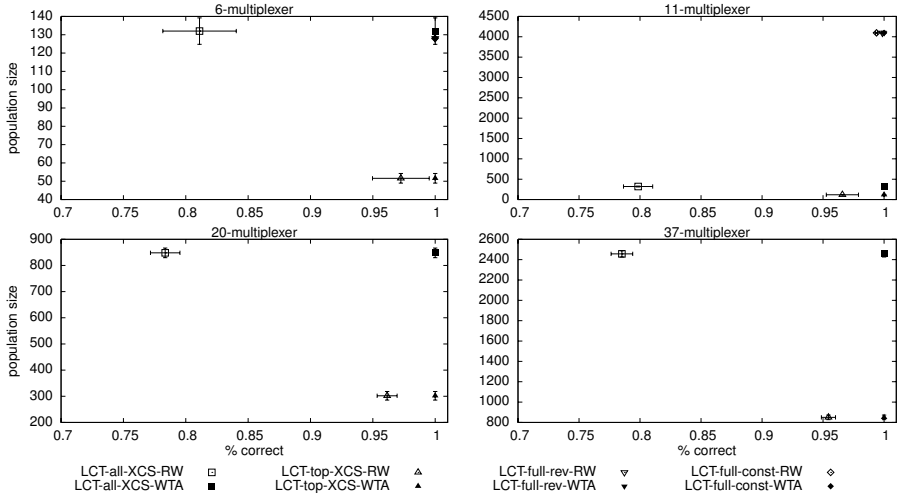

Fig. 1. Performance in the multiplexer problem. *Clockwise from upper left:* 6-, 11-, 20-, and 37-multiplexer. Within each graph, lower right is better. Note that the y-axes differ in scale. Error bars are standard deviations σ in the respective dimension.

symbols), the LCT solves the multiplexer problem perfectly , while in the original roulette-wheel (RW) strategy (empty symbols), it solves only between 80% and 97% of the problem instances. With the winner-takes-all strategy, the LCT shows the same results as the full-fledged XCS implementation presented in [5]. The figure also shows that the population size of the all-XCS translation (square symbol) is about three times the population size of the top-XCS translation (upwards triangle symbol) for all multiplexer problems. As the population sizes for the full-* translations rise exponentially, we excluded them from the 20- and 37-multiplexer problem.

All LCT configurations were able to perfectly adapt to the unexpected event of the inversed multiplexer problem (not depicted), given a rule base that the XCS has learned for the (regular, non-inversed) multiplexer problem. However, the LCT can only adapt to the inversed multiplexer problem, if the XCS was able to solve the multiplexer problem sufficiently well (e.g., because XCS' learning process was terminated prematurely). Otherwise, even if the XCS shows a correctness rate of 100%, not all LCT configurations can adapt to the inversed multiplexer. Figure 2 illustrates the case for $\overline{m}_{11}$. While the configurations all-XCS and full-const solve 80%-100% of the inversed multiplexer problem, the top-XCS and full-rev solve no more than 30%. The correctness rate did not change further until 1 million steps. We assume that the prematurely terminated XCS contains too many high-rewarding rules that are falsely marked as accurate because they were trained on only few problem instances, disturbing the results of the top-XCS and full-rev translations.

From the results in the multiplexer problem, we conclude that with the all-XCS translation the LCT shows both a high correctness rate and retains the

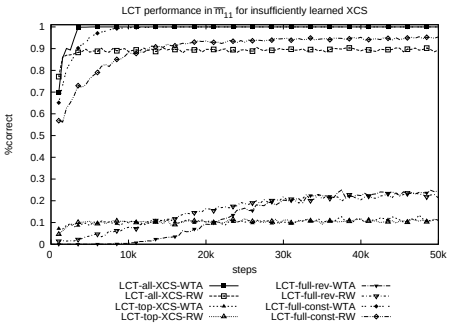

Fig. 2. LCT performance in the inversed multiplexer problem $\overline{m}_{11}$ using rules from an insufficiently learned XCS. $\sigma < 0.005$ if rate $> 80\%$; $\sigma < 0.1$ if rate $< 30\%$.

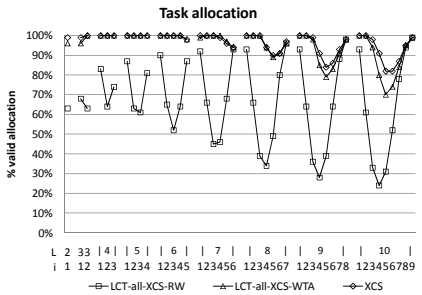

Fig. 3. Rate R_{LCT} of valid task allocations in the LCT and R_{XCS} for comparison. $\sigma < 11\%$ or better for any setting.

capability to adapt to unexpected events. When using the full-const translation, we find similar results. Combining XCS' knowledge and LCT's own learning capabilities in the full-rev translation leads to an LCT whose capability to adapt to unforeseen events is very sensitive to the quality of the XCS rules. Similar is true when using only the top performing XCS rules with the top-XCS translation. As for more real-world problem types the XCS cannot always learn perfectly, we will concentrate on the all-XCS translation in the following experiments.

6.2 Task allocation

Figure 3 shows the rate R_{LCT} of valid task allocations of the LCT for the (L, i)-task-allocation-problems, $1 \leq i < L \leq 10$, and R_{XCS} for comparison. The x-axis shows the problem instances and the y-axis shows run-time R_{LCT} and design-time R_{XCS}. From the figure we note that the LCT uses rule bases for which the XCS correctly allocates more than 90% of the problem instances for $L < 9$ and more than 80% for $9 \leq L \leq 10$, comparable to what has been reported in [2]. We find that the LCT using the winner-takes-all strategy (WTA) has very similar rates to the XCS, with a larger difference only for $L = 10$. Using the roulette-wheel strategy (RW), the LCT finds valid allocations considerably less often; in particular for $1 < i < L-1$, R_{LCT} drops as low as 22%. The reduced performance in the $(10, 5)$ and $(10, 6)$ problem instances concurs with the findings in [2] that these two problem instances are the most difficult for the XCS.

To test LCT's ability to adapt to unexpected events, we initialize the LCT with the all-XCS-translated XCS rules and let the cores fail randomly every 5 000 steps. Note that there is no further rule sharing between the XCS and the LCT besides the initialization of the LCT; we depict the XCS solely for comparison purposes.

Figure 4 shows R_{LCT} and R_{XCS} after the first (left half) and the second (right half) randomly chosen cores have failed. Note that the diagram shows fewer problem instances for the second core failure, as not every instance allows

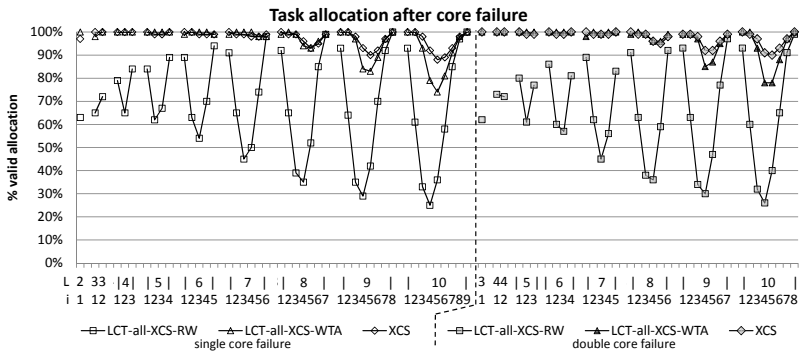

Fig. 4. R_{LCT} after one or two randomly chosen cores have failed, and R_{XCS} for comparison. After one core has failed, $\sigma < 7\%$; after two cores have failed, $\sigma < 5\%$.

the failure of two cores (e.g., when allocating three tasks out on four cores, the failure of two cores turns the problem unsolvable). We find that the rate of valid task allocations of the LCT increases slightly, on average by about 1 %-point (maximum 10 %-points) after the first core has failed and an additional 1 %-point (maximum 11 %-points) after the second core has failed. Compared to the rates before any core has failed, we find an increase of about 2 %-points on average (maximum 17 %-points). The increase is of about the same amount for any employed action selection strategy, with the roulette-wheel strategy showing a greater variance (not depicted). The results show approximately the same increase that the XCS would show. As reported in [2], the valid task-allocation rate generally increases after a core fails because the probability that the action "no valid allocation possible" is correct increases.

Summarizing, we find that when using the winner-takes-all action selection strategy, the LCT shows rates of valid task allocations which are comparable to what we find in the XCS and to what has been reported in [2]. The LCT also retains the capability to adapt to the unexpected failure of two cores, as previously shown for the XCS in [2]. The roulette-wheel strategy, however, shows high rates of valid task allocations only for some border cases.

6.3 Component parameterization

Figure 5 shows the reward returned to the LCS in the SoC component parameterization problem before (left) and after (right) the unexpected event of malfunctioning in the core, with 1000 being the maximum reward. The figure shows the results for the first 3000 steps to clearly show the reward's trend over time. We find that the less explorative winner-takes-all strategy (WTA, dashed line) receives the highest reward among the LCT configurations, with the all-XCS translation (square) being on top. While on average the roulette-wheel strategy (RW, solid line with symbols) never actively degrades performance, it is unable to achieve even the level of performance that a static, non-learning winner-takes-all strategy (cross on dashed line) achieves given the XCS-generated rule set as

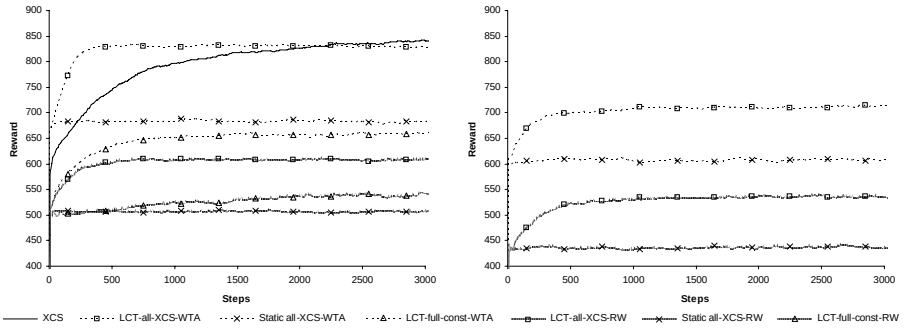

Fig. 5. Reward averaged over 100 runs for component parameterization with fully functional (left) and defective (right) component. After stabilization, $\sigma < 20$. Learning rate of XCS used to generate LCTs' initial rule set included for comparison.

a starting point. The more explorative roulette-wheel strategy is also unable to show a significantly improved learning behavior, clearly making the winner-takes-all strategy a better choice for this problem.

As expected, the initial average reward when using the full-const translation (triangle) is 500, indicating that an equal number of rules benefit and harm the system. Even though the winner-takes-all strategy is quickly able to achieve higher rewards, it is not able to achieve the same level of reward as a system initialized with a design-time generated rule set (all-XCS, square). The roulette-wheel strategy is only able to attain a very slight improvement to its average reward.

Comparing the final reward of the design-time XCS (solid line with no symbols) with the initial rewards of the run-time LCT using all-XCS translation shows a surprising discrepancy. Although the LCT uses the rules learned by the design-time XCS, we find a severe drop in the initial reward (from ∼840 to ∼650). We presume that this is because the LCT does not incorporate the complete functionality of the XCS. For example, the LCT cannot sufficiently represent XCS rules with high accuracy but low prediction, as the LCT does not store accuracy. Thus, the LCT must initially re-learn portions of the design space. Fortunately, the LCT is able to perform this initial re-learning fairly quickly within the first 500 steps.

Figure 5 shows the results of the component parameterization problem with the unexpected event as explained in Section 5. The results are very similar to those of the non-defective system, except that the average reward achieved by the system is somewhat lower than before. In fact, the starting rewards of less than 500 for the roulette-wheel strategy (solid line) indicate that, initially, a majority of actions are taken that disadvantage the system. As before, the learning capabilities of the LCT quickly achieve an increase in the average reward. However, the fact that any frequency above 100 MHz results in timing errors

prevents the system from adapting to heavier load scenarios, forcing the system to operate at a lower degree of optimality and generally reducing the achievable maximum rewards.

In summary, we find that the LCT using the winner-takes-all action selection strategy and the all-XCS translation is capable to solve the SoC component parameterization problem, even in the event of a unexpected manufacturing defect.

7 Conclusions

In this paper, we have presented a two-stage method that combines the capability of the software-based XCS with the area efficiency of the LCS hardware implementation LCT. In the first stage at design time, the XCS initially learns a set of classifiers based on a software simulation of a given problem. In the second stage, we translate the classifiers into rules that are suitable for the LCT and apply the LCT to the same problem at run time.

We showed that with our newly introduced winner-takes-all action selection strategy for the LCT, the LCT can solve the multiplexer, the task-allocation and the SoC component parameterization problem, if we initialize it with all rules that the XCS has learned (all-XCS). In addition, the LCT retains the capability to adapt to the unexpected events of the problems, which includes the unexpected failure of two cores and the manufacturing defect of a core. We also found that the performance of the LCT is less sensitive to the performance of the XCS when using the all-XCS translation.

In summary, the results show that our proposed method allows a small and lightweight yet very capable hardware implementation of an LCS, with which the autonomic control of chips using LCS becomes feasible.

In future work, we will investigate alternating between roulette-wheel and winner-takes-all action selection for quicker adaptation to unexpected events in the LCT. We will also examine ways to reflect XCS's knowledge of reward prediction accuracy in the reward of the generated LCT rules, avoiding the initial drop in returned reward, and we will look for a trade-off between the good performance of all-XCS and the smaller classifier set of top-XCS.

References

1. A. Agarwal, V. Zolotov, and D. Blaauw. Statistical clock skew analysis considering intra-die process variations. *IEEE CAD*, 23(8):1231 –1242, 2004.
2. A. Bernauer, O. Bringmann, and W. Rosenstiel. Generic self-adaptation to reduce design effort for system-on-chip. In *IEEE SASO*, pages 126–135, 2009.
3. A. Bernauer, D. Fritz, and W. Rosenstiel. Evaluation of the learning classifier system xcs for soc run-time control. In *LNI*, volume 134, pages 761–768. GI, Springer, 2008.
4. 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*, 50(4/5):433, 2006.

5. 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*, 52(8-9):516–533, 2006.
6. S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749. ACM New York, NY, USA, 2007.
7. M. Butz and S. W. Wilson. An algorithmic description of xcs. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *IWLCS '00*, number 2321 in Lecture Notes in Artificial Intelligence, pages 253–272, London, UK, 2001. Springer-Verlag.
8. M. V. Butz, D. E. Goldberg, and K. Tharakunnel. Analysis and improvement of fitness exploitation in xcs: bounding models, tournament selection, and bilateral accuracy. *Evol. Comput.*, 11(3):239–277, 2003.
9. F. Dias, A. Antunes, and A. Mota. Artificial neural networks: a review of commercial hardware. *Engineering Appl. of Artificial Intelligence*, 17(8):945–952, 2004.
10. M. Dorigo. ALECSYS and the AutonoMouse: Learning to control a real robot by distributed classifier systems. *Machine Learning*, 19(3):209–240, 1995.
11. F. Ferrandi, P. Lanzi, D. Sciuto, and D. e Inf. Mining interesting patterns from hardware-software codesign data with the learning classifier system XCS. *Evolutionary Computation*, 2:8–12, 2003.
12. J. H. Holland. Adaptation. In R. Rosen and F. M. Snell, editors, *Progress in theoretical biology*, pages 263–293, New York, 1976. Academic Press.
13. International Roadmap Committee. International technology roadmap for semiconductors. http://www.itrs.net/reports.html, 2008.
14. K. Irick, M. DeBole, V. Narayanan, and A. Gayasen. A hardware efficient support vector machine architecture for fpga. In *FCCM '08*, pages 304–305, Washington, DC, USA, 2008. IEEE Computer Society.
15. G. Lipsa, A. Herkersdorf, W. Rosenstiel, O. Bringmann, and W. Stechele. Towards a framework and a design methodology for autonomic soc. In *ICAC*, 2005.
16. V. Narayanan and Y. Xie. Reliability concerns in embedded system designs. *Computer*, 39(1):118–120, 2006.
17. H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer, and H. Schmeck. Organic control of traffic lights. *LNC*, 5060:219–233, 2008.
18. 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 *IEEE IRPS*, pages 5–10, 2003.
19. B. Widrow, D. E. Rumelhart, and M. A. Lehr. Neural networks: applications in industry, business and science. *Commun. ACM*, 37(3):93–105, 1994.
20. S. W. Wilson. Classifier systems and the animat problem. *Machine Learning*, V2(3):199–228, Nov. 1987.
21. S. W. Wilson. Zcs: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.
22. S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
23. S. W. Wilson. Generalization in the xcs classifier system. In J. R. Koza, W. Banzhaf, et al., editors, *Genetic Programming Conference*, pages 665–674, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.
24. J. Zeppenfeld, A. Bouajila, W. Stechele, and A. Herkersdorf. Learning classifier tables for autonomic systems on chip. In H.-G. Hegering, A. Lehmann, H. J. Ohlbach, and C. Scheideler, editors, *GI Jahrestagung (2)*, volume 134 of *LNI*, pages 771–778. GI, 2008.