Self-Adaptation at Chip Level

Realizing Self-Adaptation at Chip Level with Learning Classifier Systems

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Erberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Andreas G. Bernauer, M. Sc.
aus Kirchheim unter Teck

Tübingen
2011

Tag der mündlichen Qualifikation:	01.06.2011
Dekan:	Prof. Dr. rer. nat. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. rer. nat. Wolfgang Rosenstiel
2. Berichterstatter:	Prof. Dr. sc. techn. Andreas Herkersdorf

Abstract

This thesis proposes a design methodology for a self-adaptive controller to realize self-adaptation at chip level. Current system-on-chip design faces numerous problems such as process variation, transistor variability, and degradation effects, which are addressed with custom adaptive and adjustable circuits. These adaptive circuits have low design reuse rates, take a considerable amount of time during the design process, and are closely intertwined with sensors and effectors, hindering technological advancement. The proposed self-adaptive controller addresses the issues of merely adaptive circuits: it solves different adaptation problems, its design is automated by the proposed design methodology, and it connects with different sensors and effectors.

The main novelties of the proposed design methodology are employing a machine learning algorithm, namely the learning classifier system XCS, and two different versions of the self-adaptive controller at design time and run time. A machine learning algorithm reduces the design effort, as it automates the process of finding optimal parameter settings. Two different versions, a powerful but complex software version at design time and a lightweight but restricted hardware version at run time, consider the different optimization criteria of the two time periods.

Three examples and several experiments show the benefits of the proposed design methodology and the self-adaptive controller. They show how the self-adaptive controller controls the frequency and voltage of a chip, how it adapts to events that have not been foreseen during design time (such as changes in the environment or component failures), how it scales with multi-cores by cooperation without the need for central control, and how it generalizes from restricted learning at design time.

Additionally, this thesis extends the current state on XCS theory to cover problems with varying schema order, which is typically encountered in SoC adaptation problems. The thesis thus contributes to both SoC design and learning classifier systems.

Lastly, the thesis includes a simulation library, which is based on the industry standard SystemC. The simulation library co-simulates the hardware and software of the SoCs and trains the machine learning algorithm.

Zusammenfassung

Diese Dissertation stellt eine neue Entwurfsmethode für einen selbst-anpassenden Regler vor, mit der sich Selbst-Anpassung auf Chip-Ebene realisieren lässt. Der Entwurf von System-on-chip sieht sich zur Zeit zahlreichen Probleme gegenüber, etwa Schwankungen beim Herstellungsprozess, Variabilitäten in Transistoren oder Alterungserscheinungen. Diesen Problemen wird zur Zeit mit adaptiven oder einstellbaren Schaltkreisen begegnet. Diese Schaltkreise lassen sich jedoch schlecht in neuen Entwürfen wiederverwenden, erhöhen den Aufwand im Entwurfsprozess und sind eng an Sensoren und Effektoren gekoppelt, was die technologische Weiterentwicklung behindert. Der vorgestellte selbst-anpassende Regler nimmt sich der Probleme rein adaptiver Schaltkreise an: er ist in verschiedenen Einsatzbereichen anwendbar, sein Entwurf ist automatisiert mithilfe der vorgestellten Entwurfsmethode und er kann mit verschiedenen Sensoren und Effektoren betrieben werden.

Die vorgeschlagene Entwurfsmethode enthält insbesondere folgende Neuerungen: sie verwendet einen maschinellen Lernalgorithmus, nämlich das Learning-Classifier-System XCS, und zur Entwurfs- und zur Laufzeit zwei verschiedene Versionen des Lernalgorithmus. Ein maschineller Lernalgorithmus verringert den Aufwand im Entwurfsprozess, da er die Bestimmung von Kenngrößen automatisiert. Mit den zwei verschiedenen Versionen des Lernalgorithmus stehen zur Entwurfszeit eine leistungsstarke jedoch komplexe Software, zur Laufzeit hingegen eine leichtgewichtige jedoch etwas eingeschränkte Hardware zur Verfügung. Die beiden Versionen berücksichtigen dabei die jeweils unterschiedlichen Rahmenbedingungen und Optimierungsziele zur Entwurfszeit und zur Laufzeit.

Drei Anwendungsbeispiele und zahlreiche Experimente zeigen die vielseitige Einsetzbarkeit der vorgestellten Entwurfsmethode und des selbst-anpassenden Reglers. Sie zeigen, wie der selbst-anpassende Regler die Frequenz und Spannung eines Chips steuert, wie er sich an Ereignisse anpasst, die zur Entwurfszeit nicht berücksichtigt wurden (etwa Abweichungen in der Umgebung oder der Ausfall von Komponenten), wie er ohne zentrale Kontrollinstanz mit einem Multi-Core-System skaliert und wie er bei eingeschränktem Lernen zur Entwurfszeit Verallgemeinerungen trifft.

Darüber hinaus erweitert diese Dissertation die Theorie des XCS, um Problemstellungen variabler Ordnung zu lösen, die bei diesem neuartigen Einsatz des XCS auftreten. Die Dissertation enthält außerdem eine SystemC-Simulationsbibliothek, mit welcher der maschinelle Lernalgorithmus trainiert wird.

Contents

1 Introduction

This thesis proposes a design methodology for a self-adaptive controller to realize self-adaptation at chip level. The number of System-on-Chip (SoC) designs is expected to increase strongly, according to the International Technology Roadmap for Semiconductors (ITRS) (International Roadmap Committee, 2008). The major advantages of SoC designs in comparison to other designs are higher levels of system integration and reduced design costs. As in complementary metal-oxide-semiconductors (CMOS) clock frequencies cannot be increased far beyond 4 GHz due to exuberant power consumption and temperature, processing power is increased by integrating more cores and functionality on a single chip, resulting in multi-processor system-on-chip (MPSoC). High design reuse rates are necessary to keep design costs low and time to market short, despite increased functionality per chip. The ITRS estimates a requirement of a design reuse rate for SoCs of 70% until 2015 and 90% until 2020 (International Roadmap Committee, 2008, System Drivers Chapter). Design complexity is governed by the consideration of many factors. Increasing transistor variability (Bernstein et al., 2006; Borkar et al., 2003) and process variation (Agarwal et al., 2004) make it difficult to know the actual physical parameters at design time, while degradation effects (Schlunder et al., 2003) such as negative bias temperature instability (NBTI) change the physical parameters during the chip's life time. Further, reliability issues have to be considered for a successful design (Narayanan and Xie, 2006). The interaction of an increasing number of components per chip aggravates the impact of these factors on design complexity (Sander et al., 2010).

At the same time, time to market decreases. More than ever, it is important to be the first on the market, even with a subpar product that is to be improved in later versions. Hence, Rabaey and Malik (2008) find that "innovations in managing complexity and reduction in time to market are becoming increasingly more important." To achieve this goal, Borkar (2009) declares "that the future of designs in 22 nm and beyond is System Design with design automation at all levels. [We need] to research and develop capabilities for this upcoming paradigm shift." He sees the real design effort as "to optimize a system for a given purpose, with validated and well proved functional blocks, rather than to design individual logic blocks from scratch, as it is today." To meet the increasing design complexity, future design methodology thus asks for an automated process based on existing functional blocks.

Looking at design methodology, for the last 40 years, most chips have been produced based on fixed and rigid designs, where every uncertainty and dynamics about the final product and its operating conditions has been addressed with correspondingly higher design margins. However, with increasing variability, variation, and degradation, this approach becomes less favorable, as the design margins get so large that they use most of the benefits of the next technology node (Borkar, 2005; Rabaey and Malik, 2008), threatening new technology nodes to become economically untenable. To reduce the high design margins, it is necessary to reduce the underlying uncertainties and dynamics. While better physical models and fabrication technology help to reduce uncertainties to some extent, less rigid but more flexible designs are most promising. Flexible designs adapt some key parameters after design time when the actual manifestation of the uncertainty is known. For example, Mishra et al. (2009) present an adaptive controller that adjusts supply voltage to compensate for process and temperature variations. As the uncertainties and dynamics are handled at run time, designers can use smaller design margins, which makes future technology nodes economically feasible again.

Flexible designs are achieved with adaptive or adjustable circuits. Adaptive circuits adapt a run-time parameter by feedback control, while adjustable circuits allow to set some key run-time parameters after design time, by fuses, registers, memory, or similar means. For example, the low-power pipeline Razor (Ernst et al., 2003) adapts its voltage to the current temperature variations such that the timing-error rate is kept at a predetermined value. On the other hand, in ReCycle (Tiwari et al., 2007), which redistributes cycle time across a pipeline to compensate process variations, the amount of redistribution is determined with built-in self-tests and is adjusted once after production. Adaptive and adjustable circuits can also be combined for less uncertainty and more flexibility. For example, the deskew system of the Intel Itanium 2 processor, codenamed Montecito, adapts a delay buffer to nullify clock skew due to power, voltage, and temperature variations at run time (Fetzer, 2006); for maximum performance, the deskew system is also adjusted once after production to compensate clock skew due to process variation (Naffziger et al., 2005).

Generally, adaptive and adjustable circuits can be described by a sensor–controller–effector schema: a sensor senses the current state of the chip (e.g. current timing-error rate), a controller deduces an appropriate response (e.g., to increase supply voltage), and an effector carries out the response (e.g., by the dynamic voltage scaling component). The sensor–controller–effector schema is known under various names: for example, IBM calls it MAPE (monitor, analyze, plan, execute), the ASoC project uses the terms monitor, evaluator, and actuator (Lipsa et al., 2005), and the Organic Computing initiative (Schmeck, 2005) uses the terms observer and controller (Richter et al., 2006).

While current work on adaptive and adjustable circuits help to address the upcoming problems of variability, variation, and degradation in future technology nodes, it concentrates on the sensor and effector parts of the circuits, but neglect the controller part. For example, the authors of Razor mention difficulties about finding a suitable controller and settle with a proportional control system "as a starting point." Sometimes, the corresponding controller is even only envisioned, as by Karl et al. (2005), who present a circuit for aggressive voltage scaling in static random-access memory (SRAM) to "eliminate conventional design margins considering [...] process variations, local supply voltage variations, and temperature fluctuations", but do not present the necessary controller to actually achieve this goal.

Even if the controller is provided, the controller is not reusable, needs extra design effort, and, most importantly, is difficult or even impossible to extend. For example, the adaptive controller presented by Mishra et al. (2009) carefully relates the current queue length of an input FIFO to the desired processing rate and supply voltage, which cannot be easily reused in a different design and requires extra design effort. Similarly, the built-in self-tests needed to calibrate ReCycle have to be selected by the designer anew for each design. Additionally, the authors only envision that ReCycle is automatically adapting to temperature changes at run time and do not provide a workable solution. Similarly, the deskew system of Montecito is calibrated only once after production and cannot be readjusted automatically to compensate degradation effects. None of the discussed adaptive and adjustable circuits allow the inclusion of other adaptations, which limits design flexibility and prevents synergistic application of several adaptations (Hughes et al., 2001; Jayaseelan and Mitra, 2009).

1.1 Problem description

This thesis addresses the problem of a controller for adaptive and adjustable circuits that is reusable, requires little additional design effort, and is deployable with many different sensors and effectors.

As outlined above, the main problems of current controllers in adaptive and adjustable circuits are the following:

- Current adaptive circuits contain finely tuned controllers for the addressed adaptation problem. Hence, the adaptive circuits *cannot be easily reused* in other designs that address other or even similar adaptation problems.

  For example, the controller used in an adaptive circuit that tunes the clocking of pipeline stages usually cannot be used to adjust the supply voltage of an SRAM, for which a new adaptive circuit and controller have to be crafted. Even if the new adaptation problem is similar to the one for which an adaptive circuit already exists (for example, another pipeline), the existing solution has to be carefully adapted to the new adaptation problem (for example, new built-in self-tests for timing measurements). Overall, the reuse rate of current adaptive circuits and their controllers is low.

- Current controllers are usually specific solutions to individual problems. Their design *takes a considerable amount of time* during the design process and cannot be automated. Quite the contrary, the controllers require specially trained engineers who are experts in the field covered by the adaptation problem.

- The controller is often *closely intertwined with sensors and effectors*. Therefore, often neither the sensors or effectors, nor the controller can be easily exchanged for a new design. If, during the design process or the lifetime of the product, new technology makes new sensors or effectors possible, the controller cannot be easily adjusted, again requiring considerable design effort.

Given the shortcomings of current adaptive circuits, designers are forced to choose between the poor chip performance of traditional worst-case designs and the prolonged design process of designs that use current adaptive circuits. An adaptive circuit at chip level that provides good chip performance and that is reusable, that does not (overly) prolong the design process and that is deployable with different sensors and effectors is thus highly desirable.

1.2 Requirements of controllers for adaptive circuits

The shortcomings of current adaptive or adjustable circuits lead to the following requirements for adaptation at chip level:

- *Design methodology* Instead of designing specific adaptive circuits for particular problems, a design methodology for adaptive circuits is needed that can be integrated into current SoC design processes.
- *Minimum design effort* Designing an adaptive circuit should require as little effort as possible, as otherwise the benefits of better chip performance is countered by a prolonged design process.
- *High reuse rate* Not only should a particular design require a minimum design effort, but also future designs that use adaptation at chip level should require a minimum design effort to match the shrinking time to market. A major way to reduce the design effort for future designs is to ensure a high reuse rate. The controller should be easily reusable in new designs that are similar to the original one, or, even better, also in new designs that differ from the original one.
- *Feasibility* Although very sophisticated adaptation systems are easily imaginable, they have to be feasible and realizable with little additional cost and overhead, as otherwise the adaptation system would use up the benefits of the next technology node and the turn-over point of the total product cost will be reached too soon.
- *Adaptation to unexpected events* Further, given the increasing complexity of nanoscale SoCs, the controller should be able to tolerate events that have not been foreseen at design time, as ensuring that all events that negatively affect SoC performance are considered becomes more and more difficult and is time consuming. The controller should therefore be able to gracefully handle events that have not been considered at design time, that is, “unexpected” events. The important property is the *ability* to handle unexpected events—it is unrealistic to expect the controller to handle *all* unexpected events.

An adaptive system at chip level that fulfills these requirements will provide a middle ground between poor chip performance and a prolonged design process. Given these requirements, the question arises how a controller with minimum design effort, high reuse rate, and tolerance for unexpected events looks like. This leads to the proposed solution of this thesis: a self-adaptive controller.

1.3 Proposed solution: self-adaptation at chip level

The main cause for the shortcomings of current controllers of adaptive circuits are their rigidness and missing flexibility: if the controllers were more flexible, they could be reused more easily for different adaptation problems and be deployed with different sensors and effectors. An obvious solution is to make the controller more adaptive or adjustable, in the same way as it has been done with rigid designs. However, how such an adaptive or adjustable controller could look like in practice is not obvious. Further, the design effort of *how* to adapt or adjust the controller remains. The controllers are rigid because designers have to carefully manufacture them such that the controllers can correctly interpret the incoming sensor data and connect it to meaningful effector commands; it seems there was little room for flexibility. However, if the controller can learn by its own how to interpret the sensor data and what the meaningful commands are, the designer does not have to manufacture the controller anymore. Instead, the controller adapts itself to the required control task; hence, it is a *self-adapting controller*. The self-adapting controller is different from adaptive or adjustable circuits in so far as the self-adapting controller not only adapts or adjusts its subject under control, but also itself. A self-adapting controller can be reused in different designs for different adaptation problems, reduces the need for manual labor and thus the design effort, and it is deployable with different sensors and effectors.

This thesis proposes a design methodology for a self-adaptive controller to realize self-adaptation at chip level. The self-adaptive controller addresses the above mentioned problems with current controllers of adaptive circuits, while the design methodology shows how to design such a self-adaptive controller within a regular SoC design process. The thesis enables designers to realize self-adaptation at chip level with a self-adaptive controller as part of an adaptive circuit, so that designers can move forward from specific solutions to self-adaptation at chip level. The main novelties of the chip-level self-adaptation is employing a machine learning algorithm to realize the self-adaptive controller and to use different controller versions at design time and at run time, being mindful of the different optimization criteria at these time periods. For the machine learning algorithm, this thesis proposes the learning classifier system XCS, as it is a model-free reinforcement learning algorithm, as it is readily applicable using existing cost functions, allows a high reuse rate, and for which a lightweight hardware version is available, among other advantages. As the rules of the XCS are readable and writable by humans and can be exchanged between different systems, the XCS is also in line with the observation that “design complexity and time to market are causing a larger fraction of designs to be mapped on flexible and programmable platforms that can be reused over a broad application domain” (Rabaey and Malik, 2008).

The proposed self-adaptation at chip level has several benefits:

- Using a machine learning algorithm reduces the design effort as it automates the process of finding optimal parameter settings. Given an appropriate cost function or method that can asses a given chip state, the machine learning algorithm can learn how parameter changes or settings affect a given chip state and identify the optimal parameter changes or settings. The task of the designer is reduced to formalizing the assessment of a given chip state (e.g., frequency, voltage, and error rate), while analyzing the effects of parameter changes and finding the optimal parameter settings are executed by the machine learning algorithm. This falls in line with the observation that a designer usually finds it easier to assess a given chip state than to tell the parameter changes that lead towards an optimal chip state.

- Being mindful of the different optimization criteria at design time and run time allows to use a complex software version as well as a fast and efficient hardware version of the learning classifier system. While for a software version complexity is not an issue, as many computational resources are available at design time, a feasible and realistic hardware version must obey the constraints of a hardware design such as minimum area and power consumption. Instead of having similar software and hardware versions of the learning classifier system, this thesis thus proposes to use two different versions. At design time, a software version of the learning classifier system is used that is powerful but complex and computationally expensive. At run time, a hardware version of the learning classifier system can be used that is restricted but lightweight in terms of chip area and power consumption. With this, neither is the design-time representation of the self-adaptation system restricted to the capabilities of a corresponding hardware solution, nor does the run-time representation have to take much chip area or consume lots of power to match the design-time representation.

- The proposed design methodology connects the run-time system with the knowledge that is acquired during design time. The software version of the learning classifier system learns to correctly identify the optimal parameter settings for different chip conditions (e.g., an increased voltage level for increased timing-error rates) with an appropriate simulation model. The resulting set of classifiers, which represent the acquired knowledge, are transferred from their design-time representation into a form that is suitable for a run-time version of the self-adaptation system. Either the whole acquired knowledge or only a reduced subset, which is created by subsuming at the end of the design time, can be transferred to the run-time system. The transfer function ensures proper operation at run time.

- As the self-adaptive controller adapts itself to the control task, it has a high reuse rate. What changes across different problems are the number of input and output bits to and from the controller and the function or method to assess a given chip state, of which only the latter has to be redesigned for a new problem. Furthermore, as similar systems probably have similar optimal parameter settings, already acquired knowledge can be shared among similar problems; for example, the knowledge for the optimal settings of a pipeline can be used to more quickly learn the optimal settings for another similar pipeline.

- The proposed self-adaptive controller inherits the property of learning classifier systems to be applicable to a large variety of problems, from conventional chip-control problems, which require continuous adjustments, to combinatorial problems, which require finding the optimal pairing among arbitrary combinations of chip states and optimal parameter settings. This wide range of applicability also contributes to the high reuse rate of the proposed system.

- The classifiers of the learning classifier system are readable and writable by humans. If the designer already knows the optimal parameter change or setting for a given chip state, he or she can incorporate this knowledge by expressing it in terms of classifiers and including these classifiers in the self-adaptive controller. Thus, with the proposed self-adaptive controller, already taken investments into finding optimal parameter settings are not lost, or, in other words, the proposed design methodology can reuse knowledge that has been generated by means that are different from the learning classifier system. This is in contrast to other machine learning algorithms such as neural networks or support vector machines, where the stored knowledge is opaque and practically inaccessible for the designer.

- As the learning classifier system is still executing as a machine learning algorithm in its hardware version, it retains the capability to adapt to events that have not been represented or even foreseen in the simulation model, further reducing the design complexity. This ability allows the hardware version to correct inaccuracies of the design-time simulation model and reduces the design complexity.

The analysis of the proposed design methodology for a self-adaptive controller reveals further benefits and contributions to both SoC design and learning classifier systems:

- As the acquired knowledge is not opaque but readable and writable, it can be shared among several self-adaptive controllers, for example to control the cores in an MPSoC. This allows a scalable use of the self-adaptive controller in MPSoC, as shown in an MPSoC application example where cooperating self-adaptive controllers solve problems that isolated controllers could not solve.

- The thesis extends the current state of the art of distributed learning classifier systems (which the cooperating self-adaptive controllers represent): not only can they cooperatively solve a single common problem as shown previously in the literature, but they also can solve several slightly different problems, such as the control of the cores of an MPSoC with different cooling properties.

- The thesis also extends the current theory on the learning classifier system XCS. When controlling SoCs, so-called complex problems with heterogeneous schema orders are encountered, unlike regular problems that have homogeneous schema orders. The thesis extends the current XCS theory covering regular problems to also cover complex problems, such that the maximum population size and other parameters can be estimated.

- The thesis introduces a technique to transfer knowledge from one learning classifier system to another learning classifier system, which has not been done before. The experiments will show that the receiving learning classifier system benefits from this knowledge transfer. The knowledge transfer connects design-time learning with run-time learning. It covers either the whole acquired knowledge or a reduced subset that is created by subsuming after design time to save chip area in form of memory that holds the classifiers.

- The thesis suggests an alternative action selection method, namely the winner-takes-all (WTA) method, for the LCT, a possible hardware implementation of the self-adaptive controller. The thesis also provides a way to automatically generate a sensible set of rules to operate the LCT—a property that the LCT is currently lacking.

In contrast to special-purpose adaptive circuits, the proposed design methodology for a self-adaptive controller realizes self-adaptation at chip level while offering a high design reuse rate. The thesis describes the design methodology and the self-adaptive controller and refers to an appropriate version of a run-time implementation.

1.4 Structure of this thesis

The remainder of this thesis is structured as follows. Chapter 2 presents the relevant basic terms and algorithms that are necessary in this thesis. It introduces learning classifier systems and the physical models that are used for the design-time simulation models of the chip. The introduction to learning classifier systems presents learning classifier systems in general, the particular learning classifier system XCS that is used throughout this thesis, the current state of XCS theory, the multiplexer problem (a commonly used benchmark problem), and a lightweight hardware implementation of a learning classifier system, which can be used to realize the proposed self-adaptive controller.

Chapter 3 discusses work that is related to this thesis. It covers the current state of the art on self-adaptation, the Autonomic System-on-Chip project, within which this thesis has been developed, related learning classifier systems that have not been covered in the previous chapter, and current techniques on dynamic voltage and frequency scaling.

Chapter 4 presents the proposed design methodology for a self-adaptive controller in detail. First, it describes the interface of the self-adaptive controller and the design-time and run-time aspects of the design methodology, including the method of subsuming XCS classifiers after learning. Then, it describes the methods to realize self-adaptation with cooperation. The chapter introduces a formalization of the operator-allocation problem, such that it can also serve as a benchmark problem for self-adaptation systems, which current literature lacks. It continues presenting an extension of the current theory on XCS, which allows to estimate the necessary population size for complex problems at design time. The chapter ends with the SystemC simulation library that is used to co-simulate the hardware and software of the SoC and to train and simulate the XCS.

The proposed self-adaptive controller and the design methodology are applied to three application examples and evaluated in further scenarios, the experimental setups of which are described in Chapter 5. The application examples will show that the proposed self-adaptive controller can actually be used like a regular controller, that it can adapt itself to situations and events that have not been foreseen at design time, and that it is applicable without further changes to different adaptation scenarios, from traditional feedback control problems to combinatorial problems. The ability to self-adapt to unforeseen situations and events and the wide area of applicability are properties that current controllers lack.

The first application example in Chapter 5 is a simulated AMD Opteron CPU, which is a commercial quad-core multi-purpose processing unit. The experiments will show the applicability of the proposed self-adaptive controller to control a system-on-chip and to self-adapt to unforeseen events. The second application example is a multi-processor system-on-chip (MPSoC) that is based on the Cell processor. The experiments will show the scalability of the self-adaptive controller: for highly-interacting components only one self-adaptive controller per component is needed—a higher-level controller is not necessary. Additionally, the experiments show that self-adaptation with cooperation can address problems that self-adaptation that operates in isolation cannot handle. The third application example is the problem of operator-allocation. The experiments will show the wide applicability of the proposed self-adaptive controller: not only can it be applied to control problems such as in the previous problems of system-on-chip control, but it can also be applied to combinatorial problems such as the operator-allocation problem. Besides allowing a further analysis of the proposed self-adaptive controller and its ability to self-adapt to unforeseen events, the experiments will also show the possibility to reduce the necessary time for learning during design time by exploiting the generalization capabilities of the XCS.

Chapter 5 further describes the experimental setups that validate the presented extension on current XCS theory and that analyze the effects of the proposed method of subsuming after learning. The chapter ends with a section describing the experiments of two full runs of the proposed design methodology: from learning the classifiers at design time to applying the classifiers at run-time with an appropriate hardware implementation of the XCS; the application examples are the academic multiplexer problem and the newly introduced operator-allocation problem. The experiments will show that, using classifiers that have been learned during design time, a run-time hardware implementation retains the capabilities of self-adaptation, even to unexpected events.

The results of the described experiments are presented in Chapter 6, which ends with a section on the costs of the design methodology and of a possible hardware realization of the proposed self-adaptive controller. The results are discussed and summarized in Chapter 7, which concludes this thesis, describes prerequisites and limitations, and gives an outlook on future work. As references, the appendix contains the list of figures and tables, a glossary of the symbols, variables, and functions that are used in this thesis, and the bibliography.

2 Basics

This chapter presents the relevant basic terms and algorithms that are necessary in this thesis. Section 2.1 presents the learning classifier systems (LCS), in particular the learning classifier system XCS, which is used in this thesis. It also presents the current theory on learning in the XCS, the multiplexer problem, which is a typical benchmark problem, and describes the LCT, a possible lightweight hardware implementation of an LCS. Section 2.2 presents the project “Autonomic System-on-Chip”, within which this thesis has been developed. Section 2.3 introduces the physical models that are used in this thesis for the software simulation of system-on-chips (SoC). In particular, it presents the models on power consumption and temperature, as well as models on soft errors such as temperature-dependent timing errors at a transistor and radiation-induced errors in memory.

2.1 Learning classifier systems

Learning classifiers systems (LCS) are machine learning algorithms that combine reinforcement learning and genetic algorithms. They have been actively researched since the 1980s and were introduced by Holland (1976). A typical learning environment for LCS is depicted in Figure 2.1, which is typical for reinforcement learning algorithms.

An LCS is characterized by the following four main properties:

- The LCS has a set of *detectors* with which it senses its environment, like the rods and cones in a retina.
- The information processing from the detectors happens in form of *information packets*. The LCS processes these information packets and generates new packets.
- The LCS can control a set of *effectors*, which change the environment.

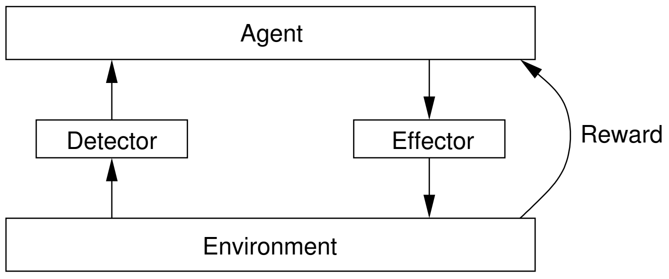


Figure 2.1: Typical setup in a reinforcement learning problem.

- The LCS receives a *reward* (a *feedback*) from the environment that indicates how "good" the LCS' reaction has been.

A learning classifier system mainly solves three problems in the realm of machine learning (Holland et al., 2000): interaction and coordination of rules, credit assignment and discovery of new rules.

Interaction and coordination of rules A machine learning algorithm cannot possibly know every possible combination of perceptions in advance (e.g., "a man with a red coat walks in the street"). However, if the machine learning algorithm knows the building blocks (e.g., "man", "red coat", "street"), it can handle the situation quite easily.

Combining building blocks or rules allows a machine learning algorithm to handle many new situations, and because the combinations are so manifold, the machine learning algorithm can handle many different situations. Moreover, because useful building blocks appear frequently, the machine learning algorithm can test and validate them thoroughly. The problem lies in how the machine learning algorithm accomplishes the interaction and coordination of a large number of simultaneously active rules.

The LCS approaches this problem of rule interaction and coordination by restricting the rules on emitting information packets. The information packets may contain *tags*, which can be easily used as an address and to coordinate actions. Thus, the LCS is nothing more than a system that processes an active list of information packets. As the information packets just trigger rules, the LCS avoids the problem keeping rules without contradiction: the active list of information packets just results in a list of active rules.

Credit assignment A great problem with rule-based machine learning algorithms is the question which rule contributed to the success of the algorithm. This is particularly difficult if only a certain sequence of actions lead to a desired outcome (e.g., sacrificing a pawn in the game of chess).

In real environments, it is impossible to test all possible sequences of actions. The LCS tries to solve that problem by establishing a market situation: every rule acts as a broker in a chain of rules starting from the current situation to the (possibly) desired outcome. In every link every rule with a matching condition bets that it gets activated. If the rule gets actually activated, it pays its bet to the rules preceding it in the chain (its suppliers), because these rules allowed it to get activated. The activated rule in turn collects the bets of the succeeding rules (consumers) that get activated and accumulates capital (the *rule strength*). The last consumer is the environment, that rewards the LCS for its actions.

Discovery of new rules For rule-based machine learning algorithms, discovering new rules is a great yet still unfathomed problem. It is obvious that the machine learning algorithm must replace unhelpful rules. However, it is not obvious by what to replace them. Random rules are helpful for small problems. For large problems, the key to success lies in exploiting the accumulated knowledge and derive plausible hypothesis or rules about the current situation (without introducing overt contradictions).

LCSs generate new rules with a genetic algorithm and use its ability to recombine existing building blocks. The genetic algorithm operates on both kinds of building blocks that exist in LCS: the building blocks of the rules (condition and action) and the “building blocks” of the chain of rules that lead to a certain action. The genetic algorithm uses the rule strength as the fitness value, which thus affects both building block levels.

Every described mechanism is supposed to enable the LCS to constantly adapt to its environment and meet the environment’s challenges step by step. In the course of adaptation, the LCS constantly strives to balance *exploration* (acquiring new information) and *exploitation* (efficiently using acquired information).

LCS problems

The application of LCS results by some problems, which are mentioned by Smith et al. (2000) and outlined here briefly.

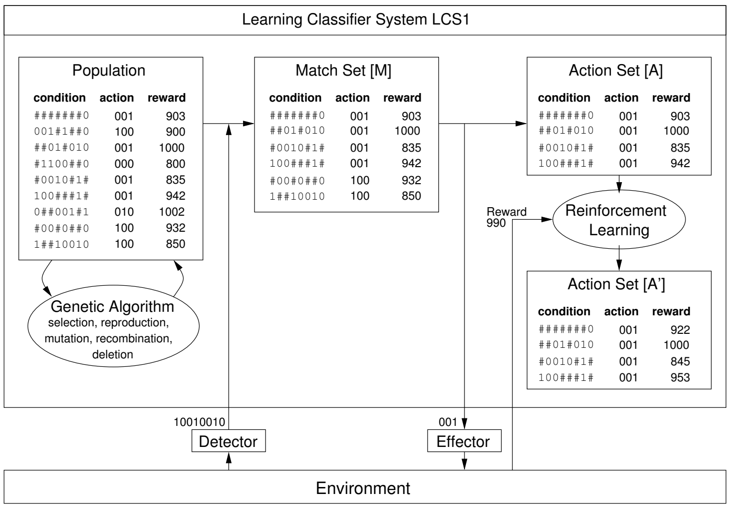
Some problems originate from LCS usually being described as an algorithm, but actually rather being a strategy. When applying the LCS, you have to determine how to implement the strategy. Among others, the following questions arise:

- What is a good representation of the environment within the LCS?
- How do you distribute the reward among the rules?
- How can you change the internal information-packet processing such that the LCS can learn in environments that lack the Markov property, that is, the property that the optimal action only depends on the current situation, as in the game of chess, not on previous actions, as in the game of skat.

Additionally, once you have implemented the LCS, you can observe the following phenomena, all of which have a detrimental effect on the learning process:

- Sometimes some strong rules can suppress many weak rules, although the weak rules also contribute to the success of the LCS.
- This problem is often connected to the problem of credit assignment: who in the chain of rules shall earn how much from the reward?
- Rules that are very general (i.e., rules that contain many don’t-care symbols) can overly suppress other, more specialized rules.
- The attempt to adapt the internal information-packet processing to learn in environments without the Markov property seems to aggravate the problem and up to now only leads to parasitic rules. Parasitic rules regularly receive a reward because they place themselves into the chain of rules, but they reduce the performance of the system.

The learning classifier system XCS, which is explained later in Section 2.1.2, addresses most of these problems. It represents the environment by binary strings of ‘0’ and ‘1’ and a don’t-care symbol (‘#’) to match a portion of the state space and it distributes the reward by Q-learning (Watkins, 1989). The XCS addresses the problem of strong or general rules suppressing weak or specialized rules by selecting classifiers according to


Learning Classifier System LCS1
Genetic Algorithm
selection, reproduction, mutation, recombination, deletion
Reinforcement Learning
Reward 990
10010010
Detector
001
Effector
Environment

Population

condition	action	reward
#######0	001	903
001#1##0	100	900
##01#010	001	1000
#1100##0	000	800
#0010#1#	001	835
100###1#	001	942
0##001#1	010	1002
#00#0##0	100	932
1##10010	100	850

Match Set [M]

condition	action	reward
#######0	001	903
##01#010	001	1000
#0010#1#	001	835
100###1#	001	942
#00#0##0	100	932
1##10010	100	850

Action Set [A]

condition	action	reward
#######0	001	903
##01#010	001	1000
#0010#1#	001	835
100###1#	001	942

Action Set [A’]

condition	action	reward
#######0	001	922
##01#010	001	1000
#0010#1#	001	845
100###1#	001	953

Figure 2.2: Structure of the typical learning classifier system LCS1. Adapted from Butz et al. (2006).

their accuracy in predicting the reward. However, the XCS still relies on environments that have the Markov property.

Before introducing the XCS, a more formal description of a simple learning classifier system, the LCS1, follows.

2.1.1 A simple learning classifier system: LCS1

This description follows the presentation of the LCS1 of Butz (2006). The LCS1 highlights the major aspects of a modern version of an LCS as it was envisioned by Holland and it allows to show the differences and advantages of the XCS over traditional LCS. The LCS1 is a Michigan-type LCS, where the LCS has only a single rule set. It is the dominant type of LCS and first exemplified with CS-1 by Holland and Reitman (1978). The alternative Pittsburgh-type LCS maintains a set of rule sets. It is originally developed in LS-1 (Smith, 1983) and is not detailed further here.

Figure 2.2 shows the structure of the LCS1. The LCS1 contains a set of rules in a *population*. The rules consist of a *condition*, *action*, and an (expected) reward or *rule strength*. The rules are also called *classifiers* and are denoted by cl. The condition $cl.c$ of the rule is encoded in binary, so that a genetic algorithm can operate on it. The condition's length is denoted by L. The special symbol '#' is a don't-care symbol and matches both '0' and '1'. The *specificity* $\sigma(cl)$ of a classifier is the ratio of the number of '0' and '1' to the number of '#'. It indicates how much of the state space the classifier covers. A classifier with k '0' and '1' has a specificity of $^k/_l$.

For every problem instance s ('10010010' in Figure 2.2), the LCS1 chooses the rules from the population whose condition matches the problem instance (sensing by the detector). These rules constitute the *match set* $[M]$. From the match set, the LCS1 selects an action with a P_{explt}-greedy action selection policy: with probability P_{explt} the

LCS1 exploits the already acquired knowledge and selects the action whose classifiers expect the highest relative reward:

$$\pi_{\mathrm{LCS1}}(s) = \begin{cases} \arg\max_a \frac{\sum_{cl \in [M]|_a} cl.r}{\|[M]|_a\|} & \text{with prob. } P_{\mathrm{explt}} \\ \mathrm{random(A)} & \text{otherwise} \end{cases} \tag{2.1}$$

where $[M]|_a = \{cl \in [M] \mid cl.a = a\}$ is the set of all classifiers in the match set that propose action a, $cl.r$ is the expected reward of the classifier, and A is the set of all possible actions. When the LCS1 selects $a = \pi_{\mathrm{LCS1}}(s)$ as the action ('001' in the figure), it forms the *action set* $[A] = [M]|_a$.

After receiving reward R for applying action a ($R = 990$ in the figure) and forming the match set $[M]^+$ in the subsequent problem instance s^+, the LCS1 updates the reward prediction of the classifiers in $[A]$ according to the modified Q-learning equation (Watkins and Dayan, 1992)

$$cl.r \leftarrow cl.r + \beta \left(R + \gamma \cdot \max_a \frac{\sum_{cl' \in [M]^+|_a} cl'.r}{\|[M]^+|_a\|} - cl.r \right) \tag{2.2}$$

which estimates the maximum future reward from the average of all participating classifiers, using the learning rate β and the reward discount factor γ. Unlike Q-learning, the LCS1 uses a set of classifiers instead of a single classifier to estimate the future reward. If all classifiers were maximum specific (i.e., $\sigma(cl) = 1$ for all cl), the LCS1 would perform Q-learning.

The LCS1 generates new rules with two methods: during *covering* and with a *genetic algorithm* (GA). Covering happens if no classifier matches the problem instance, mostly at the beginning of learning when the population is empty. During covering, the LCS1 creates a classifier whose condition matches the current problem instance s except for a random number of locations that it replaces by '#' with probability $P_{\#}$. The action of the newly created classifier is random. The genetic algorithm is in the simplest form a steady-state (or incremental) genetic algorithm (Whitley and Kauth, 1988). Depending on the expected reward, the genetic algorithm selects two rules from the population, mutates it at random locations, recombines them with crossover and adds them to the population. Two other rules are selected randomly according to the inverse of their expected reward and are deleted from the population.

2.1.2 The learning classifier system XCS

The learning classifier system XCS was proposed by Wilson (1995) and is a milestone in the development of learning classifier systems. It differs from other learning classifier systems mainly in the following way (see also Figure 2.3):

- The fitness of a rule in the genetic algorithm does not depend on the rule strength, but on the accuracy of the reward prediction. With this, rules that do not contribute high rewards but predict this fact accurately, can remain in the population.

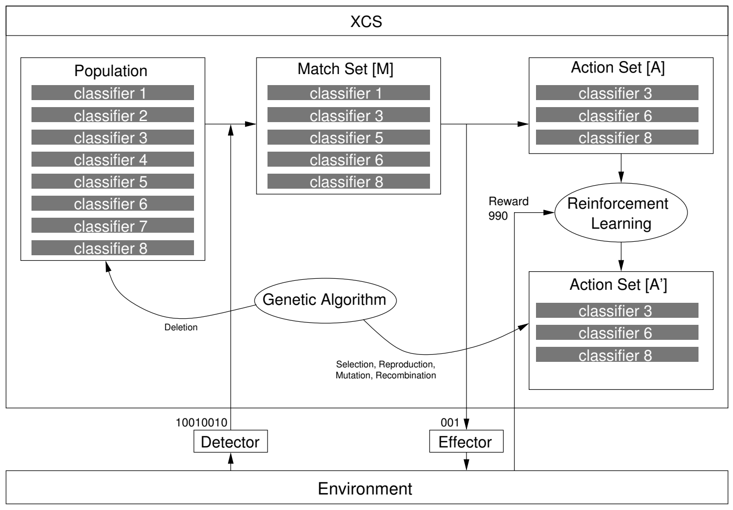


Figure 2.3: Learning in the XCS. Adapted from Butz et al. (2006).

- The genetic algorithm only operates on the match set or the action set but deletes rules in the whole population (*panmictic deletion*). With this, the XCS learns in niches, as only the rules that match a certain situation are considered for generating new rules.

Wilson (1995) shows that the XCS learns the mapping $(C, A) \Rightarrow P$ from the set of all pairs of conditions (C) and actions (A) into the set of rewards (P) completely and accurately. Furthermore, the XCS tends to keep the rules as general as possible. These findings were confirmed in a more detailed analysis by Kovacs (1997). Thus, the XCS fulfills two important properties of an LCS: complete and accurate rules that are as general as possible.

Formally, the classifier of an XCS consists of five main parts plus some others. The five main parts are:

- The *condition* c of a fixed length L specifies in which situations the classifier is applicable.
- The *action* a from a fixed set $A = \{a_1, \ldots, a_n\}$ specifies the proposed action if the condition is met.
- The *prediction* r specifies the expected reward if the action gets applied under the given condition.
- The *prediction error* ε specifies the average absolute error of the reward prediction.
- The *fitness* f specifies the accuracy (the inverse error) of the classifier relative to the other classifiers of the same niche (i.e., a subspace of the current problem).

The condition c, the action a and the prediction r have the same syntax and semantics as in LCS1. The XCS uses the prediction error ε to prefer classifiers that accurately predict the reward and are thus excellent specialists in their niche. On the other hand, the XCS uses the fitness f in the genetic algorithm so that only rules that actually are rivals (as they apply in the same niche) do compete with each other.

Additionally to the five main parts, the following are also parts of an XCS classifier:

- The *action set size estimate as* specifies the running average of the size of the action set of which the classifier was a part.
- The *time stamp ts* specifies the last time (in steps) in which the classifier took part in the genetic algorithm.
- The *experience exp* specifies how often the parameters of the classifier have been updated.
- The *numerosity num* specifies the number of micro classifiers that the classifier represents: a single macro classifier can cover several micro classifiers.

The parameter update in the XCS works similar as in LCS1, with the following differences:

- After forming the match set $[M]$, the XCS forms a *prediction array* $P(a)$ that contains the expected reward for every possible action:

  $$P(a) = \begin{cases} \frac{\sum_{cl \in [M]|_a} cl.r \cdot cl.f}{\sum_{cl \in [M]|_a} cl.f} & \text{if } [M]|_a \neq \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

  $P(a)$ is thus the fitness-weighted average of all predicted rewards in $[M]$ that propose action a. The XCS uses the prediction array to select an appropriate action according to some selection strategy. The most common selection strategies are random selection (usually used in the explore mode), winner-takes-all selection, and roulette-wheel selection.

  - *Random selection* selects the action at random with a uniform distribution (the prediction array is of no use in that case, of course).
  - *Winner-takes-all selection* selects the action

    $$a_{\max} = \arg\max_a P(a) \tag{2.4}$$

    that expects the highest reward.
  - *Roulette-wheel selection* selects the action with a probability proportional to the entry in the prediction array: $\Pr(a) = {}^{P(a)}/_{\sum_a P(a)}$.

- The XCS updates its parameters usually in the following sequence: prediction error ε, prediction r, and fitness f. If $P^{t+1}(a)$ is the prediction array of the next step, then let $Q = \max_{a \in A} P^{t+1}(a)$.

- The prediction error ε of a classifier cl in $[A]$ is then:

  $$cl.\epsilon \leftarrow cl.\epsilon + \beta(|\rho - cl.r| - cl.\epsilon) \tag{2.5}$$

where in classification problems $\rho = R$, the received reward, and in reinforcement problems $\rho = R + \gamma Q$. β and γ have the same meaning as in LCS1 (learning rate and reward discount factor).

- The reward prediction r is calculated as:

$$cl.r \leftarrow cl.r + \beta(\rho - cl.r) \tag{2.6}$$

  The XCS performs thus Q-learning that is based on all classifiers in $[A]$.

- The fitness update of a classifier in $[A]$ depends on its relative accuracy κ' that is derived from the prediction error ε as follows:

$$cl.\kappa = \begin{cases} 1 & \text{if } cl.\varepsilon < \varepsilon_0 \\ \alpha \left(\frac{cl.\varepsilon}{\varepsilon_0}\right)^{-\nu} & \text{otherwise} \end{cases} \tag{2.7}$$

$$cl.\kappa' = \frac{cl.\kappa \cdot cl.num}{\sum_{cl' \in [A]} cl'.\kappa \cdot cl'.num} \tag{2.8}$$

  Here, $cl.\kappa$ is the current absolute accuracy of the classifier and $cl.\kappa'$ the relative numerosity-weighted accuracy. It uses a power function with the exponent ν to prefer classifiers with a small error. Classifiers with an error $\varepsilon < \varepsilon_0$ are considered accurate.

  The fitness is updated according to:

$$cl.f \leftarrow cl.f + \beta(cl.\kappa' - cl.f) \tag{2.9}$$

- The action set size as is updated according to the same β scheme:

$$cl.as \leftarrow cl.as + \beta(|[A]| - cl.as) \tag{2.10}$$

- Every time the XCS updates the parameters of a classifier, it increases the classifier's experience by one.

- The XCS updates the parameters $cl.r$, $cl.\varepsilon$ and $cl.as$ in the described way only if the classifiers experience $cl.exp$ is larger than $1/\beta$ (*moyenne adaptive modifiée* (MAM) presented by Venturini (1994)). Otherwise, it uses the average values of the first $cl.exp$ runs to speed up convergence of the initial system.

- After updating the prediction, the prediction error and the action set size estimate, the XCS may perform *action set subsumption* (ASS). It searches the action set for the most general classifier that is accurate ($cl.\varepsilon < \varepsilon_0$) and sufficiently experienced ($cl.exp > \theta_{\text{sub}}$). Then, it deletes each classifier of the action set from the population that is covered by the most general classifier, whose numerosity $cl.num$ is then increased by one. The general classifier is said to *subsume* the other classifier. The idea is that every value that the covered classifier adds to the whole system is already accomplished by the most general classifier.

The genetic algorithm also differs from the genetic algorithm in LCS1. In the XCS version of Wilson (1998) that is used in this thesis, it operates on the action set.

- The genetic algorithm operates only on the current action set, if at least θ_{GA} steps have passed since last time.
- The genetic algorithm selects the parent classifiers with a probability proportional to their relative fitness:

  $$\Pr(cl \text{ is selected}) = \frac{cl.f}{\sum_{cl' \in [A]} cl'.f} \tag{2.11}$$

  The parents remain in $[A]$ and thus compete with their offsprings.
- The genetic algorithm generates the offsprings by copying the respective parent p. Crossover happens with probability χ_{GA}; in that case, the prediction, fitness and prediction error of the offsprings are the averages of the respective parent values, otherwise just the value of the parent. Furthermore, for each offspring $cl.f \leftarrow {}^{p.f}/_{p.num}$, where p is the original parent, and $cl.exp \leftarrow cl.num \leftarrow 1$. The genetic algorithm mutates the offsprings' conditions with probability μ_{GA} either arbitrarily or so that it more closely matches the current situation (*niche mutation*).
- Before the genetic algorithm adds the offsprings to the population, it checks whether the offsprings are covered by a sufficiently accurate ($cl.\varepsilon < \varepsilon_0$) and sufficiently experienced ($cl.exp > \theta_{\mathrm{sub}}$) parent. If this is the case, the genetic algorithm does not add the offspring but instead increases the numerosity of the parent (*GA subsumption*). The reasoning is the same as for action set subsumption: every value that the offspring may add to the whole system is assumed to already be accomplished by the parent.
- If the population exceeds its maximum population size N by the added offsprings, the genetic algorithm deletes classifiers, each with a probability proportional to $cl.as$. If a classifier is experienced enough ($cl.exp > \theta_{\mathrm{del}}$), but its fitness is significantly worse than the average $\bar{f}$ of the population ($cl.f < \delta\bar{f}$, where δ is an arbitrary parameter usually set to 0.1), the genetic algorithm increases the classifier's deletion probability by ${}^{\bar{f}}/_{f}$.

The reproduction of new classifiers in $[A]$ by the genetic algorithm and the deletion of classifiers in $[P]$ results in optimal classifiers as proposed in Wilson's Generality Hypothesis (Wilson, 1995) and theoretically justified by Butz et al. (2004b). Optimal classifiers are complete, accurate and maximally general classifiers: they cover all possible input states, accurately predict the reward for the optimal action and realize a population of minimal size. The Generality Hypothesis also states that while the two subsumption methods GA subsumption and action set subsumption help to stronger condensate the population, they are not necessary to evolve accurate classifiers. Besides, the resulting populations are more vulnerable to changing environments such as the environments considered in this thesis, which also exhibit unforeseen events.

2.1.3 XCS theory

Naturally, the evolution of optimal classifiers depends on the settings of the various parameters of the XCS. Recently, some theoretic work has been developed that helps in setting the parameters of the XCS to ease the evolution of optimal classifiers. Butz et al. (2004b) identify the *covering challenge* and the *schema challenge* that the parameter settings must meet to allow the evolution of optimal classifiers. Butz et al. (2003) describe how the parameters influence the *reproductive opportunity* of the classifiers. In the following, the theory of these three theoretic boundaries are laid out. Note that even if the parameter are not selected optimally, intrinsic fitness guidance and a biased reward function can still help to evolve optimal classifiers.

Covering Challenge

When an input state is not covered and N classifiers already exist, the XCS must delete an existing classifier to make room for a new covering classifier. If this occurs too often, the XCS deletes classifiers before they can accumulate enough experience to have meaningful fitness values. Without meaningful fitness values, there is no fitness pressure in the GA and the XCS cannot evolve optimal classifiers. Hence, to avoid too many deletions, each input state must be matched by at least one classifier after the population is filled up. In other words, the probability Pr(cover) that at least one classifier covers a given input state must be high enough.

The probability Pr(cover) that at least one classifier covers a given input state depends on the probability Pr(match) that a random classifier matches a given input state. The classifier matches a given input state if, for each bit in the input state, the corresponding classifier bit is specified and matches the input state bit or it is the don't-care symbol. Given the average proportion of specified bits $s([P])$ of the population, the so-called *specificity*, Pr(match) is thus:

$$\begin{aligned}\Pr(\text{match}) &= \left(\frac{1}{2}s([P]) + (1 - s([P]))\right)^L \\ &= \left(\frac{2 - s([P])}{2}\right)^L\end{aligned} \tag{2.12}$$

With this,

$$\begin{aligned}\Pr(\text{cover}) &= 1 - \Pr(\text{no match in [P]})\\ &= 1 - (1 - \Pr(\text{match}))^N\\ &= 1 - \left(1 - \left(\frac{2 - s([P])}{2}\right)^L\right)^N\end{aligned} \tag{2.13}$$

At the beginning, the classifiers will be created at random. Therefore, the specificity $s([P])$ of the filled up population is initially $s([P]) = 1 - P_\#$, and $\Pr(\text{cover})$ is

$$\Pr(\text{cover}) = 1 - \left(1 - \left(\frac{1 + P_\#}{2}\right)^L\right)^N \tag{2.14}$$

Because the XCS meets the covering challenge if $\Pr(\text{cover})$ is high, the equation shows that the XCS meets the covering challenge if $P_\#$ and N are high. Note that there are several simplifications involved in the derivation of Equation (2.14). The equation assumes a uniform distribution of the input states and ignores several effects and parameters that affect the fitness of a classifier, such as MAM, the learning rate β, or other parameters such as ε_0, ν, or α. Because of these assumptions and simplifications, Equation (2.14) is a rather rough estimate. However, Butz et al. (2004b) shows that the equation is sufficient to approximately estimate settings of $P_\#$ and N that allow the evolution of optimal classifiers.

Schema Challenge

Experienced fitness values are not sufficient though to evolve optimal classifiers—the fitness values also have to be accurate. Accurate fitness values are achieved by sufficiently specific classifiers that have as few don't-care symbols as possible. Of course, the necessary number of specified bits is highly dependent on the problem, and usually there must be sufficient fitness pressure to converge the population towards accurate fitness values. If there is no fitness guidance, the problem of converging towards accurate fitness values is known as the *schema challenge.*

Let us assume that the given environmental niche requires k specified bits to be covered accurately. This number is called *schema order* or *problem dimension* (Goldberg, 1989). A classifier represents an environmental niche if it proposes the action of the environmental niche and correctly specifies all k positions. The probability that a single classifier correctly represents an environmental niche is thus

$$\Pr(\text{one representative}) = \frac{1}{n}\left(\frac{s([P])}{2}\right)^k \tag{2.15}$$

where n is the number of possible actions and $s([P])$ the average proportion of specified bits in the population, as mentioned previously. Assuming a uniform distribution of the input states as for the covering challenge, $s([P]) = 1 - P_\#$, and the probability that the population contains at least one classifier that correctly represents an environmental niche becomes

$$\Pr(\text{representative}) = 1 - \left(1 - \frac{1}{n}\left(\frac{1-P_\#}{2}\right)^k\right)^N \tag{2.16}$$

The equation shows that, in general, setting $P_\#$ low and N high enough meets the schema challenge and allows the evolution of accurate classifiers. Note that the covering and the schema challenge ask for different settings for $P_\#$. Note also, that the equation covers the worst case, which is the general case with no fitness guidance.

Many problems provide some form of fitness guidance that eases the schema challenge. Two typical forms of fitness guidance are *layered payoff* and *biased generality.* Layered payoff provides more than two reward values to guide the overgeneral classifiers towards more specific and accurate classifiers, in particular when the degree of class affiliation is known. Biased generality is present when overgeneral classifiers are implicitly more often correct than wrong or vice versa. Changes of the GA in the overgeneral classifiers will thus change its statistical correctness and eventually lead towards more accurate classifiers. For further details on how layered payoff and biased generality ease the schema challenge, see Butz et al. (2004b).

Reproductive Opportunity

Meeting the covering and schema challenges ensure meaningful and accurate fitness values. For classifiers with meaningful and accurate fitness values to survive in the population, it must be ensured that they can reproduce before they get deleted: they must have a *reproductive opportunity.* As the GA operates on the action set, a classifier must be in the action set to actually be able to reproduce (which will be the case if the classifier is accurate). Thus, to ensure a reproductive opportunity, the probability of a classifier to be part of an action set must be higher than the probability of being deleted. The probability of a classifier *cl* with specificity $\sigma(cl)$ to be part of an action set is

$$\Pr(cl \text{ in } [\mathrm{A}]) = \frac{1}{n}\left(\frac{1}{2}\right)^{L\cdot\sigma(cl)} \tag{2.17}$$

Assuming there is no influence by fitness or action set size estimates, the probability of deletion is random and thus

$$\Pr(\text{deletion}) = \frac{1}{N} \tag{2.18}$$

As two classifiers are deleted from the population, the reproductive opportunity is met if $\Pr(cl \text{ in } [\mathrm{A}]) > 2\Pr(\text{deletion})$:

$$\frac{1}{n}\left(\frac{1}{2}\right)^{L\cdot\sigma(cl)} > \frac{2}{N} \tag{2.19}$$

Table 2.1: Population specificity $s([P])$ in theory and from empirical results. From Butz et al. (2003).

μ_{GA}	0.02	0.04	0.06	0.08	0.10	0.12	0.14	0.16	0.18	0.20
$s([P])$, theory	0.040	0.078	0.116	0.153	0.188	0.223	0.256	0.288	0.318	0.347
$s([P])$, empirical	0.053	0.100	0.160	0.240	0.287	0.310	0.329	0.350	0.367	0.394

The expected average specificity of a classifier that specifies k positions can be derived from the average proportion of specified bits $s([P])$ in the population (k specified bits individually and $L - k$ bits specified as in the population):

$$E(\sigma(cl \text{ with } k \text{ specified bits})) = \frac{k + (L-k)s([P])}{L}. \tag{2.20}$$

Once meaningful and accurate classifiers in the population are established, the average proportion of specified bits $s([P])$ in the population of course starts to deviate from its initial value for random populations of $1 - P_{\#}$. In that case, $s([P])$ depends on the mutation rate μ_{GA}. Although Butz et al. (2003) give a formula for the relation between $s([P])$ and μ_{GA}, they also note that the empirical values are slightly higher than the theoretical values because of higher noise in more specific classifiers and the fitness-based deletion method. Table 2.1 compares the empirical values with the theoretical values for various values of μ_{GA}.

2.1.4 The multiplexer problem

A typical benchmark problem to assess the learning abilities of an LCS is the *multiplexer problem* (Wilson, 1998). In the n-multiplexer problem, the LCS is expected to return the value of a particular bit in the input as an action. Given an input bit string of length $n = l + 2^l, l \in \mathbb{N}$, the first l bits index a bit in the remaining 2^l bits. The LCS is expected to return the value of the indexed bit. For example, in the 6-multiplexer problem, $m_6(\underline{01}11\underline{0}1) = 0$ and in the 11-multiplexer problem, $m_{11}(\underline{011}00001\underline{1}000) = 1$. Butz et al. (2004b) show that the XCS is able to solve the 70-multiplexer problem, which involves $2^{70} \simeq 10^{21}$ possible input strings.

2.1.5 Implementation of an LCS in hardware: the LCT

Zeppenfeld et al. (2008) introduces a lightweight hardware implementation of a learning classifier system, called Learning Classifier Table (LCT). The LCT borrows from the XCS and one of its predecessor LCS, the strength-based ZCS (Wilson, 1994), to realize a fast and efficient machine-learning building block. The LCT allows to realize CPUs with learning capabilities, so that the CPUs can react to various error situations under different SoC operating conditions in a flexible way.

Similar to other LCS implementations, the LCT maintains a list of rules or classifiers (compare Figure 2.4). Each classifier consists of a condition, an action, and a fitness.

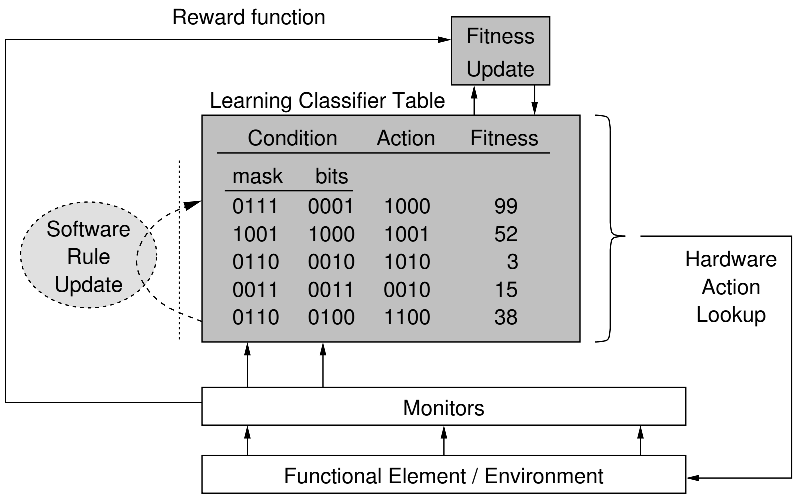


Figure 2.4: Structure of the Learning Classifier Table (LCT). Adapted from Zeppenfeld et al. (2008).

To realize the don't-care symbol, the condition is internally stored with a mask and a value, where the mask's zero bits are used to silence the don't-care positions in the input string. The fitness is similar to the strength value in the ZCS and resembles the classifier's expectation for a high reward. This is in contrast to the fitness value in the XCS, which represents the relative reward-prediction accuracy.

The learning cycle is similar to the learning cycle in the XCS. Given an input state, the LCT identifies the matching classifiers and uses roulette-wheel selection to determine the action to be executed. Once the action has been executed, the LCT receives a reward from the environment and updates the fitness of the triggered classifier. The learning cycle is single-pass weighted reservoir sampling (Efraimidis and Spirakis, 2006; Vitter, 1985) with a reservoir size of one. This avoids the calculation of a prediction array.

For the sake of a lightweight and efficient hardware implementation, the LCT does not implement any LCS components that change the list of rules, that is, there is no coverage, deletion, reproduction, mutation, or combination of rules. In particular, the initial rule set is manually crafted. These components are supposed to be realized by a future external software implementation. However, rule deletion can be mimicked by setting the fitness value of the rule to zero. The LCT learns solely by fitness update.

A preliminary synthesis of the rule selection algorithm of the LCT on a Xilinx Virtex-II field-programmable gate array (FPGA) needs 38 slices, 45 flip-flops, 59 LUTs, one multiplier, and one block random-access memory (RAM) (Zeppenfeld et al., 2008). The rules are stored in the block RAM, which can hold up to 512 rules. The lookup of a single rule takes about 7.6 ns (i.e., the LCT can run on about 131 MHz), so that a complete read–write cycle through the list of rules takes about 10 μs.

Zeppenfeld et al. (2008) simulate a system that consists of two processing cores, a shared bus, a memory, and an input–output interface and that is governed by the LCT. The LCT can keep the utilization of the processing cores near a predetermined level of 50% under changing workload. Zeppenfeld and Herkersdorf (2010) extend the work and use the LCT to equally distribute the load across the processing cores of a system that forwards Internet Protocol (IP) packets. The simulated system consists of three processing cores, a shared bus, a memory, an Ethernet medium-access controller (MAC), and an interrupt controller. Each processing core uses its own LCT. Monitors register the frequency, utilization, and workload of the processing cores; actuators allow frequency adjustment and task migration. The workload information is shared among the three LCTs so that each LCT can determine its share of the overall workload. Sending bursts of 1000 packets with a fixed inter-arrival rate of $4\,\mu$s and a total length of 4 ms, the authors show that the system can maintain operation within about 10% of the optimum. The authors also find that their results are comparable to a run-to-completion system that uses dynamic voltage and frequency scaling (DVFS).

2.2 Autonomic System-on-Chip

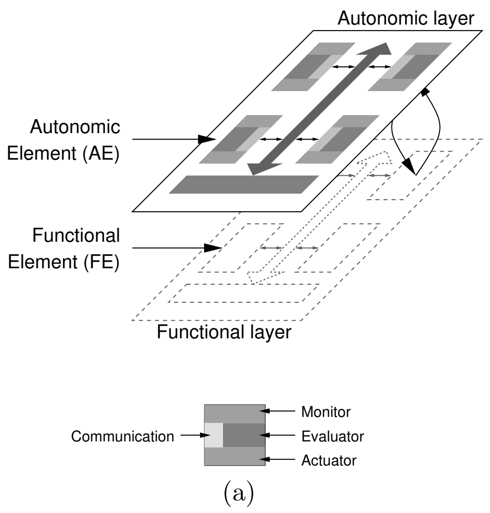
This thesis has been developed within the Autonomic System-on-Chip (ASoC) project. The ASoC project addresses the complexity and reliability problems that current and future SoC designs face, and aims to solve these problems by applying autonomic and organic principles.

The project aims to create a hardware architecture that integrates flexibility and learning capabilities in order to deal with failures, changing environments and varying workloads. Additionally, it develops a corresponding design methodology that is aware of the hardware's abilities and is able to handle reliability as a primary optimization criterion along with the traditional objectives of performance, power and area. The chip will use its flexibility and ability to learn not only to react on failures, reduced performance or increased power usage, but also to anticipate upcoming events, to build fallback scenarios and to act proactively. Flexibility and learning go together, as flexibility without learning would merely increase the complexity while learning without flexibility would be unhelpful. Flexibility and learning also mean that the hardware must make decisions during run time that have previously been decided at design time. This is another reason for a new design methodology, as it must be able to delegate some decisions into run time. Making decisions at run time has many benefits. As the process variations increase with decreasing feature size, designing for the worst case is likely to result in decreased yield, while designing for the average case may not be applicable for a particular chip. However, if the chip can adapt to its final hardware realization and make former design decisions during run time, it can compensate for specific defects or process variations unique to that chip.

As the hardware has to run on its own, it has to make decisions autonomously. For this, the ASoC project receives inspiration from the way nature deals with complexity: subsystems such as muscles or the nervous system run autonomously up to a certain point and optimize themselves. For example, the vegetative nervous system ensures the proper functioning of vital parts of the body, nerve cells used for seeing in the brain are rededicated to new functions in case of blindness, and muscles contract more synchronously when they are trained to generate a stronger force. In analogy, the ASoC project extends the functional layer of a system-on-chip (SoC) by an autonomic layer of subsystems which optimizes the functional layer and ensures that the system's vital components remain operational. The autonomic layer also allows for graceful degradation if it cannot ensure full operation. This new kind of SoC is called Autonomic System-on-Chip (ASoC). As this is a shift in the way of thinking about SoCs, the ASoC project takes an evolutionary approach: it first tries to leave the functional layer untouched by simply adding an autonomic layer. This avoids a complete redesign of the system and makes it possible to reuse existing functional elements and preserve the huge investments that have already been made into various libraries of intellectual property (IP). Once the benefits of the ASoC approach has been demonstrated, future SoC components can already be designed with the autonomic layer in mind.

The ASoC project subsumes the properties of self-healing, self-configuration, self-optimization and so on as self-x properties, as in the Organic Computing initiative (Müller-Schloer et al., 2004). There are many projects that use organic concepts to deal with complexity, all of which act on the box level (i.e., with complete computer systems). In contrast to these projects, the ASoC project targets the chip level (i.e., the hardware basis of the systems at the box level). The ASoC project believes that it is both necessary and profitable to investigate organic concepts at the chip level, similarly to how operating systems are tailored to the hardware they are running on. Hardware can handle failures within few clock cycles while software will need thousands of clock cycles; even worse, a hardware failure may prevent the software from running at all. The investigations of the ASoC project shows the benefits of the support for organic concepts in hardware and what the interface to the running software may look like.

The ASoC project proposes to rededicate a fraction of the abundant chip capacity of future SoCs to implement organic concepts. Figure 2.5a shows the envisioned ASoC architecture platform. The platform consists of two logic layers, the functional layer and the autonomic layer. The functional layer with its functional elements (FE) corresponds to what is used in today's SoCs. The autonomic layer with its autonomic elements (AE) implements the organic concepts and is one of the two main contributions of the ASoC project. The logic separation into two layers helps to separate the functional part of a (current) SoC and the new autonomic part of the chip. Physically, the two layers will be intertwined on a single die. The autonomic elements of the autonomic layer consist of four parts: a monitor that collects status information from the supervised functional element, an evaluator that evaluates and analyzes this status information and determines any necessary action, an actuator that performs this action, and a communicator that allows interaction between multiple AEs. The status information of an ASoC is manifold: it includes inputs from temperature sensors, counts of observed errors, measurements of performance and power, information on defective or degraded units and much more. As the evaluator analyzes and makes local decisions based on this status information, the interplay between the AEs will show emergent behavior which is investigated in the ASoC project. Possible actions of the actuator include stalling the CPU pipeline, initiating a micro-rollback to fix certain errors, or to induce the migration of tasks between various CPUs in multi-processor SoCs.



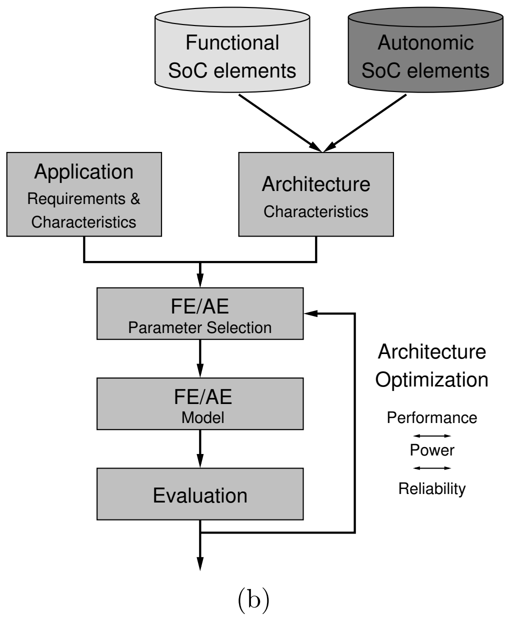

Functional SoC elements
Autonomic SoC elements
Application Requirements & Characteristics
Architecture Characteristics
FE/AE Parameter Selection
FE/AE Model
Evaluation
Architecture Optimization
Performance
Power
Reliability
(b)

Figure 2.5: (a) Architecture and (b) design methodology of an Autonomic System-on-Chip (ASoC). The ASoC logically adds an autonomic layer to the functional layer of a current SoC design with little modifications. The design flow combines functional and autonomic aspects of the design, optimizing the architecture for run-time performance, power consumption, and reliability.

The other main contribution of the ASoC project besides the two-layered architecture is a design methodology that is aware of reliability issues along with the new abilities of the ASoC target platform. As outlined above, reliability issues will get more important with decreasing feature sizes in future SoCs. The ability of the new design methodology to model and analyze the reliability and fault tolerant behavior of SoCs is critical for future SoCs, as it allows finding the optimal trade-off between the reliability, performance, power, and area of a chip. Current design methodologies do not take reliability issues as a design parameter into account (as they do with performance, power, and area), since reliability issues were either of little concern with existing geometries, or the reliability was increased after the design phase by adding protective shields (as in satellites) or triplicating the complete hardware (as in airplanes). The design flow of the new methodology is based on today's well-established Y-flow as depicted in Figure 2.5b. The ASoC project differentiates between generic AEs that can be inserted into the ASoC as a separate IP (e.g., a reliability estimator), and specific AE components that can only be added as extensions to FEs (e.g., shadow registers).

Besides the new architecture platform and design methodology, the ASoC project develops new on-chip supervision and validation techniques. Functional test coverage of multi-hundred million transistor macros cannot be exhaustive, so the new techniques will evaluate the dynamic behavior of SoC macros during run time. The trade-off between reliability, performance and power will not be considered only during design time, but also at run time. The ASoC project investigates and develops concepts for the dynamic management of these three optimization goals. It also investigates and develops mechanisms for hardware-software repartitioning via an appropriate interface, such that software can replace permanently failed units. A simulation model of the system allows to evaluate the dynamic behavior of all the various autonomic and functional components. The result will be a self-organized SoC that delivers its functionality in a dependable, fault-free way while ensuring optimal performance.

SystemC-based simulation libraries

The ASoC project uses SystemC (SystemC)[1] to simulate the SoC at design time. SystemC is a standardized ANSI-C++ class library to write software simulations of systems consisting of hardware and software. It is used by many industrial companies. It is intended for designers who wish to explore the design of their hardware-software system and its alternatives at an early stage using only a software simulation, which allows short evaluation cycles and thus saves costs. For this, the SystemC library provides classes and methods to model the system at various time scales, from untimed transaction level (Cai and Gaijski, 2003) to clock-cycle–accurate register-transfer level, including mixtures of the time scales.

In SystemC, both the architecture of the system and the implementation of the system modules are described in the C++ program. This is unfortunate for external programs that wish to only manipulate the architecture of the system, as it is difficult to extract the architecture from the C++ program or change it in place. As the ASoC projects wants to explore the addition and placement of autonomic elements, whose implementation is already fixed, it is affected by this design choice of the SystemC standard. To overcome this restriction, the ASoC project has developed the libslim library, to which this thesis contributed parts of it (see Section 4.5). The libslim library allows to separate the architecture of a system from the implementation of its modules. While the implementation is still described in the C++ program, the architecture is described in an external configuration file. The configuration file currently uses the extensible markup language (XML), so that external programs can easily discover the architecture and manipulate it. The configuration file describes the modules of the system, their interconnection, and the parameters of the modules, if any. libslim reads the architecture configuration file at the start up of the simulation, creates the described module hierarchy and starts the SystemC simulation. An additional benefit of libslim is that changes in the architecture do not require a recompilation of the simulation software.

On top of libslim, the ASoC project developed the libasoc library, to which this thesis contributed parts of it, too (see Section 4.5). The libasoc library provides modules that represent functional and autonomic elements, such as CPUs, power monitors, and the XCS evaluator.

[1] An implementation of SystemC is freely available at `http://www.systemc.org`.

2.3 Physical models

As the XCS is trained at design time with a software simulation of the final design, an accurate simulation of the physical properties of the final design is crucial. This section presents current physical models of power consumption, temperature, and (soft) error rates as used in the experiments to train the XCS at design time and to simulate the run time with differently behaving environments or additional unexpected events. In the experiments of this thesis, the performance is measured with the frequency as a (rough) estimate; later setups can include more sophisticated performance measures. As the mean time to failure due to hard errors is usually in the scale of several years, the effect of hard errors is not modeled.

2.3.1 Power consumption

The total power consumption P_{total} consists of the static power dissipation P_s and the dynamic power dissipation P_d:

$$P_{\text{total}} = P_s + P_d \tag{2.21}$$

The static power dissipation P_s is caused by leakage currents at the transistor. It is modeled according to Butts and Sohi (2000) with

$$P_s = V_{\text{DD}} N k_{\text{design}} \hat{I}_{\text{leak}} \tag{2.22}$$

where V_{DD} is the supply voltage of the transistor, N are the number of transistors, and k_{design} and $\hat{I}_{\text{leak}}$ are design- and technology-dependent parameters, which are given by Butts and Sohi (2000).

The dynamic power P_d is needed to load transistors and capacities. It is modeled according to the model presented by Weste and Eshraghian (1993) and as done by Intel to estimate power dissipation in the Pentium M (Genossar and Shamir, 2003). The model uses a so-called activity factor α that estimates the average number of zero-to-one transitions during a clock cycle (the transitions that actually draw current). The activity factor can be gained through logic simulation. The model to estimate the dynamic power dissipation is

$$P_d = \alpha C_L V_{\text{DD}}^2 f \tag{2.23}$$

where C_L is the lump capacitance and f is the clock frequency.

2.3.2 Temperature

The temperature of a device mainly depends on the power consumption of its building blocks and their relative positions. This thesis uses the HotSpot tool presented by Skadron et al. (2004) to simulate temperature distribution across the SoC. The HotSpot tool uses the well-known duality between heat transfer and electrical circuits: both systems are governed by the same set of differential equations. Heat flow between two points is modeled as a current through an electrical circuit, with the resulting voltage representing temperature difference. Thermal capacitance and resistance are modeled with their electrical counterparts, leading to electrical R-C constants representing thermal R-C constants.

The HotSpot tool describes the SoC as a connected set of building blocks, each of which has an individual power dissipation. The heat goes through a heat spreader to a heat sink, which cools the device through air convection. The HotSpot tool has been validated against a thermal test chip (Huang et al., 2004) and an FPGA (Velusamy et al., 2005).

The tool comes with a set of reasonable default values of physical constants, which have not been altered in this thesis unless stated otherwise.

2.3.3 Soft errors

Soft errors are run-time errors that happen intermittently, that is, they happen without permanently damaging the chip. The soft errors considered in this thesis are timing errors and soft errors in memory.

Temperature-dependent timing errors

Timing errors happen if the clock period is smaller than the time that a signal needs to stabilize at a transistor. This happens, for example, if the signal run times do not match the clock frequency due to aging of some components. Defining an accurate model for timing errors is difficult, as the timing error depends on the actual path the signal is taking. Therefore, a simple model is used where a fixed set of inverters between the two pipeline stages are assumed, modeling either the longest or the average path length. The average switching time of an inverter is modeled according to the temperature-dependent model of Golda and Kos (2003):

$$t_{\mathrm{av}} = \frac{t_{\mathrm{r}} + t_{\mathrm{f}}}{4} + t_{\mathrm{dT}}(T) \tag{2.24}$$

t_{r} and t_{f} are the rise and fall time delays of a signal at an inverter (Weste and Eshraghian, 1993) while t_{dT} models the influence of temperature on the time delay.

The rise time delay t_{r} is defined as the time delay that the output signal of an inverter needs to rise from 10% to 90% of its stationary value, while the fall time delay t_{f} is the time delay that the output signal of a inverter needs to fall from 90% to 10% of its stationary value. They are calculated as

$$t_{\mathrm{f}} = 2\frac{C_{\mathrm{L}}}{\beta_{\mathrm{n}} V_{\mathrm{DD}}(1 - V_{\mathrm{n}})}\left[\frac{V_{\mathrm{n}} - 0.1}{1 - V_{\mathrm{n}}} + \frac{1}{2}\ln(19 - 20V_{\mathrm{n}})\right] \tag{2.25}$$

and

$$t_{\mathrm{r}} = 2\frac{C_{\mathrm{L}}}{\beta_{\mathrm{p}} V_{\mathrm{DD}}(1 - V_{\mathrm{p}})}\left[\frac{V_{\mathrm{p}} - 0.1}{1 - V_{\mathrm{p}}} + \frac{1}{2}\ln(19 - 20V_{\mathrm{p}})\right] \tag{2.26}$$

with

$$V_{\mathrm{n}} = \frac{V_{\mathrm{tn}}}{V_{\mathrm{DD}}} \quad \text{and} \quad V_{\mathrm{p}} = \frac{V_{\mathrm{tp}}}{V_{\mathrm{DD}}} \ . \tag{2.27}$$

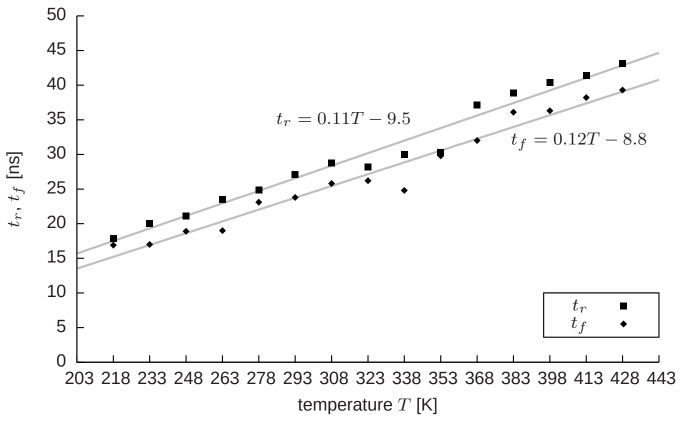


Figure 2.6: Temperature dependency of the rise (t_r) and fall (t_f) time at an inverter. Adapted from Golda and Kos (2003).

Here, C_L is the lump capacitance of the transistor, β_n and β_p are the gains of the n-type and p-type metal-oxide-silicon transistors (NMOS and PMOS), respectively, and V_tn and V_tp are the respective threshold voltages, all of which depend on process parameters and the device geometry.

The influence of temperature on the time delay is modeled according to the following linear model as shown by Golda and Kos (2003)

$$t_\mathrm{dT}(T) = \frac{T - 218\,\mathrm{K}}{428\,\mathrm{K} - 218\,\mathrm{K}} \cdot t_\mathrm{g} \tag{2.28}$$

where t_g is the total difference of the signal delay at 218 K and 428 K (see Figure 2.6).

Soft errors in memory

The effect of frequency and voltage scaling on fault rates in memory is modeled using a Poisson distribution with parameter

$$\lambda = \lambda_0 10^{\frac{(1-f)d}{1-f_\mathrm{min}}} \tag{2.29}$$

according to Zhu (2006), where d is a constant (usually less than five) and λ_0 is the average fault rate corresponding to the maximum voltage V_max and maximum frequency f_max.

3 Related work

This chapter describes work that is related to this thesis, most notably self-adaptation and (distributed) learning classifier systems. Section 3.1 presents the current state of the art of self-adaptation related with software and hardware systems. The section gives an overview of the Autonomic and Organic Computing initiatives, which deal with self-adaptive computer systems, it gives a short overview on emergence, it presents the current state of the art of self-adaptation in hardware, and it discusses definitions of self-organization. Section 3.2 presents work on learning classifier systems that is related to this thesis. The section first presents work that uses learning classifier systems for robot control, continues presenting the only known hardware implementation of XCS, and ends with presenting current work on distributed LCS. The chapter ends with Section 3.3, which presents one of the most elaborated fields in dynamic adaptations at chip level, namely dynamic adaptations to reduce power consumption.

3.1 Self-Adaptation

The idea that systems organize themselves and adapt themselves to the environment in which they are running and to the needs of their users, is very old. One of the earliest publication concerning self-adaptation might be the publication of Ashby (1947) on his principles of self-organization, which are presented later in this section. However, as the presented works will show, self-adaptation at the chip level has not been investigated so far, as until now, the (perceived) costs for self-adaptation (e.g., chip area or power) have been higher than the expected utility, the balance of which has changed with the advent of nano-scale SoCs.

This section starts with today's initiatives that address self-adaptation in computer systems, namely the Autonomic Computing and the Organic Computing initiatives. It then presents the tightly coupled topic of emergence before reviewing current work on self-adaptive hardware and finishing with the broader topic of self-organization.

3.1.1 Autonomic and Organic Computing

One of the most recent initiatives for self-adaptation in computer systems are Autonomic Computing and Organic Computing. In 2001, the computer manufacturer's IBM senior vice president of research, Paul Horn, identified the complexity of computer systems as the next "Grand Challenge" of information technology. In a manifesto[1], he stated that

[1] `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf` (retrieved January 7, 2011).

there would not be enough skilled people to keep the world's computing systems running due to its sheer complexity. He asked for autonomic systems that can administrate themselves, inspired by the autonomic nervous system of humans that regulate numerous essential and non-essential activities such as heart beat, blood pressure, digestion, or eye accommodation. Today's companies have to integrate several heterogeneous computer systems, which even extend into the Internet. Thus, not only the single software system but also the concert of all interacting software has to be considered for autonomic computing, which is seen as the only remaining option to get hold of the increasing complexity of today's computer systems. Kephart and Chess (2003) details the aspects of self-management that should be addressed in autonomic computing and ultimately become emergent properties of a general architecture: self-configuration, self-optimization, self-healing, and self-protection. Self-configuring systems seamlessly set their individual parameters to follow high-level policies and integrate heterogeneous systems. Self-optimizing systems continually try to improve their own performance and efficiency. Self-healing systems automatically repair detected software and hardware problems. Self-protecting systems defend themselves against malicious attacks and emit warnings in anticipation of system-wide failures.

Parashar and Hariri (2005) lists several systems that incorporate autonomic properties. OceanStore (Kubiatowicz et al., 2000) is a global, highly-scalable data storage that addresses all aspects of self-management mentioned previously. It builds upon an infrastructure of untrusted servers while still guaranteeing persistence of the stored data. Océano (Appleby et al., 2001) can manage computing resources for e-business software farms in a scalable way. It allows for flexible service-level agreements with the shop owners and covers peak loads which are an order of magnitude greater than the usual load. SMART DB2 (Lohman and Lightstone, 2002) reduces the cost and necessary amount of human interaction for the DB2 database system.

Organic Computing (Müller-Schloer, 2004; Schmeck, 2005) heads towards a similar direction as Autonomic Computing but stresses more the needs of humans and the lifelike properties of the resulting systems, in particular concerning ubiquitous embedded systems. This is in contrast to Autonomic Computing that seems to be more concerned with the technical realization of large-scale software systems. Schmeck et al. (2010) identifies some major properties of organic computing systems and presents a road map to ideal Organic Computing systems. The German Research Foundation (DFG) sponsors several research projects in a special priority program on Organic Computing, which covers wide and heterogeneous application areas:

The "Marching Pixels" project applies organic principles to realize self-organizing smart pixel sensors (Fey and Schmidt, 2005). The "Marching Pixels" are objects that travel and self-organize on a pixel array to solve tasks such as object, object center point, and trajectory detection. Dittrich (2005) and Matsumaru and Dittrich (2006) use the metaphor of bio-chemical information processing as a programming paradigm for distributed systems, following organic computing principles. The AutoNomos project (Wegener et al., 2006) uses organic-computing methods to develop a distributed, self-organizing traffic information system. It uses decentralized car-to-car communication to sample and report traffic information, trying to avoid the costly centralized traffic information systems. The

Table 3.1: Types of emergence. Adapted from Fromm (2005).

Type	Name	Roles	Predictability	System	Example
I	Intentional	Fixed	Predictable	Closed, passive entities	Clock
II	Weak	Flexible	By simulation	Open, active entities	Swarm
III	Multiple	Fluctuating	Chaotic	Open, multiple levels	Game of life
IV	Strong	New world of roles	Not predictable	New or many systems	Real life

Orca project (Mösch et al., 2006) develop a software architecture for autonomous mobile robots that includes a self-organizing component. Implemented into a six-leg robot, the self-healing controller can keep the robot walking after the amputation of up to two legs, without being explicitly programmed for leg amputations (El Sayed Auf et al., 2006).

The OTC project (Rochner et al., 2006) develops an organic traffic control system for urban road networks. The controller consist of two layers for on-line and off-line parameter optimization. The layer for on-line parameter optimization uses a modified real-valued LCS to select appropriate parameters for the traffic-light controller, based on the current traffic-flow data. If no classifier of the LCS matches the current traffic-flow data, the layer for off-line parameter optimization evolves a new set of parameters with an evolutionary algorithm and a simulation model. The two-layer approach results in a controller that self-adapts to the current traffic, continuously optimizing the traffic flow (Prothmann et al., 2008).

While the idea of autonomic or organic computing system is wide-spread and applied to many application areas, so far no research has been conducted on autonomic or organic computing at the chip level. To the contrary, current research is concerned with large to very large systems, neglecting the underlying basis on which all systems are built upon, the microchips. Furthermore, there is hardly any research on using learning classifier systems to realize self-adaptation, with OTC being the only known exception.

3.1.2 Emergence

Emergent behavior is a phenomenon intrinsically tied to OC principles. While there is currently no commonly accepted definition of what exactly emergent behavior means (Fromm, 2005; Heylighen, 1991), there seems to be a common understanding of what the necessary preconditions or properties of emergent behavior are.

Emergent behavior requires a multi-layered system view. System components at the lower layer (functional elements or building blocks) behave according to component-specific local rules. The local interaction between the components leads to an aggregate behavior of all constituents at the (higher) system level. This global system behavior cannot be predicted by observing the constituents in isolation.

In Fromm (2005), the author describes and classifies four different types of emergence, summarized in Table 3.1. Type I is the intentional type of emergence: there is no feedback, the roles of the components are fixed and the system behaves as designed and predicted like a clock. Type II is a weak form of emergence with feedback and flexible components. The behavior cannot be predicted anymore, but simulated like for a swarm of birds. Fromm calls type III emergence multiple emergence, which has many feedbacks and shows a chaotic behavior similar to John Conway's game of life. It is hard to assign certain roles to the components as they vary over a large range. Finally, type IV emergence is the strongest form of emergence like we know it from real life. Roles are continuously generated and deleted on many different layers and nothing is predictable anymore. Other authors have presented similar classifications of emergence (Bar-Yam, 2004).

The self-adaptive controller presented in this thesis exhibits emergence at least of type II, as it receives feedback with the reward from the environment and realizes a flexible component. When several self-adaptive controllers cooperate with each other, the transmitted classifiers also transmit feedback that other controllers have received, which may lead to type III emergence. However, the roles assigned to the controllers are not as fluctuating as required in type III emergence, so that the proposed self-adaptive controller has to be placed as a strong type II emergence component with a flavor of type III.

3.1.3 Self-adaptive hardware

While there are adaptive and adjustable hardware circuits, actual *self-adaptive* hardware has not been realized so far, the difference being that an adaptive hardware circuit adapts some *other* part of the hardware, while a self-adaptive hardware circuit also adapts *itself*.

There exist many adaptive and adjustable hardware circuits, which are too numerous to list them all. Examples are given in the introduction: Mishra et al. (2009) present an adaptive controller that adjusts supply voltage to compensate for process and temperature variations; the low-power pipeline Razor (Ernst et al., 2003) adapts its voltage to the current temperature variation such that the timing-error rate is kept at a predetermined value; the ReCycle circuit (Tiwari et al., 2007) redistributes cycle time across pipeline stages to compensate process variation; the Intel Itanium 2 processor (Montecito) has an active and adjustable deskew system to nullify clock skew due to process, power, voltage, and temperature variations (Naffziger et al., 2005), a power monitor to throttle instruction execution if too much power is consumed, and a cache monitor that removes defective memory cells to prevent unrecoverable errors (Fetzer, 2006). In all examples the adaptive or adjustable circuit adapt some other part of the system—the adaptive circuit itself is not adapted.

What comes closest to the proposed self-adaptive controller are (other) machine learning algorithms that are implemented in hardware. The most popular machine learning algorithms for which hardware implementations exist are neural networks (Dias et al., 2004; Widrow et al., 1994) and, more recently, support vector machines (Anguita et al., 1998, 2003; Irick et al., 2008). Along with the fact that for these systems, "the actual rules implemented [are] not apparent" (Widrow et al., 1994), their implementations

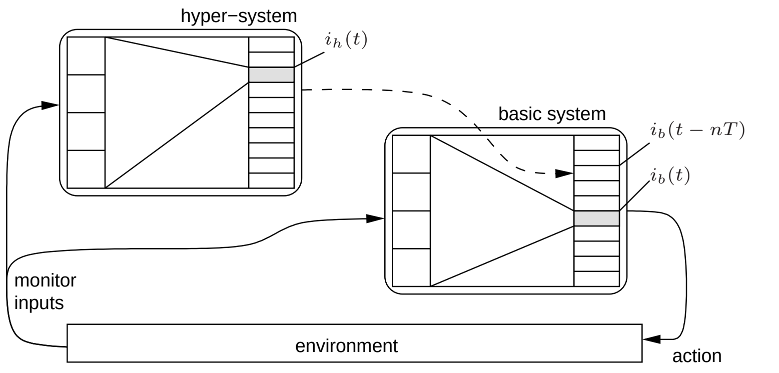


Figure 3.1: Structure of the self-learning Aron system. Adapted from Brockmann (1992).

are about five times as large as the LCT (Zeppenfeld et al., 2008), mostly due to the involved kernel functions. Further, these hardware implementations are intended to improve execution speed of the machine learning algorithms but not to adapt a SoC or similar hardware.

A general engineering technique to realize the functional mapping of inputs to outputs are lookup tables (LUT), which is used in numerous adaptive circuits (e.g., Mishra et al. (2009)). Usually, LUTs only store the output values and use the input value as an index into the table. For example, to avoid calculating $sin(x)$, a memory can hold at each position x the result of $sin(x)$, given a sufficient discretization of x. Instead of using x directly as the address into the memory, some kind of hash value of x can be used instead, for example if the domain of x is sparse. While LUTs allow an adaptive behavior of a system, their major disadvantage is that they are static: if the system deviates from the behavior that has been supposed at design time, there is no self-adaptation of the LUT itself.

The Aron system (short for “Alternatives Regularly Organized and Numbered”) presented by Brockmann (1992) aims to overcome the disadvantage of LUTs being static by stacking two or more LUTs into a hierarchy (see Figure 3.1). Each LUT uses a hash function to translate feature values (e.g., “very high speed” and “medium voltage”) into a so-called instance number and to summarize these instance numbers into a so-called situation number by weighted addition. The situation number is used to index the lookup table that contains the action to be performed (e.g., “increase voltage by 0.5 V”). The lookup table at the bottom of the hierarchy is called basic system, while the lookup table above the basic system is called hyper-system. The basic system is similar to the LUT described previously and outputs appropriate actions for given input values. The hyper-system adjusts the basic system if the system under control deviates from an implicitly given ideal goal, generally by adding the action value of the hyper-system to the action value of the basic system. All lookup tables are initialized by an expert of the system under control. While the Aron system overcomes the disadvantage of LUTs of being static, it requires the expert of the system under control “to recognize the functional relations between suboptimal results and the modifications of the basic [system].” In particular, the expert must know both the optimum of the system under control and how to reach this optimum from any suboptimal operating point. Furthermore, it is unclear how the hyper-system shall change the basic system if the necessary change is to use some other parameter *combination* instead of just adjusting a single parameter (e.g., "increase voltage by 0.5 V and set frequency to 100 MHz"), as the hyper-system is implicitly supposed to be able to adjust the basic system repeatedly.

3.1.4 Self-Organization

A topic that is related to emergence and particularly to organic computing is self-organization. Scientists have investigated self-organization since the beginning of cybernetics in the 1940s. William Ross Ashby published his principles on self-organization in 1947 (Ashby, 1947). He defeated the view of self-organization being something additional to the parts of a system. Instead, he defined organization as the state transition function of a finite state machine. From this definition follows that there cannot exist a machine S that really self-organizes (as the state transition function cannot be a function of the state of the machine, otherwise this would result in just another state transition function), but there must exist an external machine A that changes the state transition function of the self-organizing machine S, that is machine S is only self-organizing in the environment of $S + A$ (Ashby, 1962). Ashby also stresses the importance of the observer as it is the observer who identifies the parts of a (real) system and their interactions.

Later works use similar definitions for self-organizing systems as finite state machines whose transition function changes and where the observer takes an important role. In Lendaris (1964), the author defines a self-organizing system as a finite state machine whose state transition function varies over time. However, he does not mention who varies the state transition function. In Heylighen (2004), the author also describes the properties of self-organized systems. He sees the need for a model with which the self-organized system evaluates its actions (a prerequisite to changing the state transition function). But as the author sees this to "rather fit the paradigm of centralized control", he does not pursue this aspect further. Gershenson and Heylighen (2003) and Herrmann et al. (2006) are examples of recent works which stress the importance of the observer. In their view, it depends on the observer's description of the system whether the system can be considered self-organized.

Recent authors measure the degree of self-organization in a system with Shannon's information theory, for example Guerin and Kunkle (2004), Parunak and Brueckner (2001), and Herrmann et al. (2006). Unlike Ashby, they relate organization to the order or disorder of the system in terms of its Shannon entropy S

$$S(n) = -\sum_{i=1}^{n} p_i \ln p_i \tag{3.1}$$

where n is the number of available states and p_i is the probability to reach state i.

Although useful, this definition has two caveats: (i) entropy is an extensive function (i.e., it increases with the size of a system) and (ii) entropy and order can increase together. As entropy depends on the available number of states n, the entropy of a system increases with the size of the system, although the degree of order is (intuitively) the same (e.g., compare one liter of water to two liter of water). Dividing by the amount of possible states in terms of bits overcomes this disadvantage (Landsberg, 1978, p. 366):

$$D(n) = \frac{S(n)}{\ln n} = \frac{-\sum_{i=1}^{n} p_i \ln p_i}{\ln n} \tag{3.2}$$

With this, the disorder D lies between 0 and 1 and order can be defined as $1 - D$. If n can vary over time, Landsberg (1984) showed that entropy and order can increase at the same time, as $S(n)$ can increase less rapidly than $\ln n$. However, if n is constant, the definition of order in terms of entropy complies with the intuitive understanding.

Herrmann et al. (2006) attempt a more formal approach to define self-organizing systems. According to the authors, a self-organizing system must adapt by changing its own structure (which constrains the system's constituents) and must not have a central control (which would be a single point of failure). While the criterion for adaptation sounds reasonable, the criterion of having no central controller does not. The authors state that a central controller would turn any system self-organizing, as long as its description includes the central controller. However, this is the same reason why the authors stress the importance of the description of the system. Up to this day, it remains open whether this formal approach proves to be vital. In the meantime, a system containing the proposed self-adaptive controller falls within the definition: it changes the constraints on the system components so that they adapt to failing components and it does not need a central supervisor.

3.2 Learning classifier systems

This section covers work on LCS that is related with hardware or that exploits the population-based nature of LCS to distribute the classification task. LCS are used both for control problems and for classification problems, of which the latter is the most common use case. This section first presents work on LCS to control robots, which face similar problems as the proposed self-adaptive controller. In a similar spirit, Studley addresses the semi-Markov decision problem of distributing rewards to the classifiers of an LCS while the LCS controls a robot by proposing an event-based LCS named X-TCS, which this section presents next. The section continues with presenting the only known attempt on a hardware implementation of XCS and ends with presenting work on distributed LCS.

While there is existing work on LCS to control computing hardware, in every case the LCS is a software process that controls a hardware; no work uses LCS to realize self-adaptive hardware and no previous work is known in this respect. Additionally, no work presents a methodology where the classifiers are learned beforehand instead of directly on the device. Further, the distributed LCS instances cooperate to solve a single task, which differs from the additional use case proposed in this thesis, where distributed LCS instances cooperate to solve several, (slightly) different tasks.

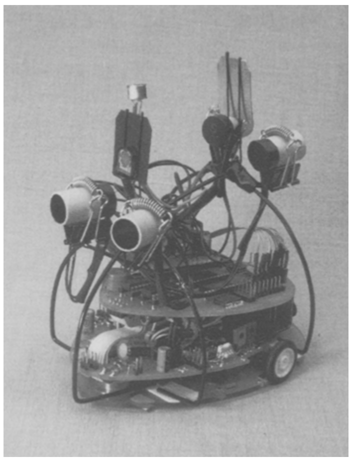


Figure 3.2: The AutonoMouse. From Dorigo (1995).

3.2.1 AutonoMouse

Doriga and Schnepf (1993) propose a hierarchy of learning classifier systems to control a simulated robot. The simulated robot has sensors to detect light, heat, food, and predators, depending on the task, and can move in eight directions. The investigated hierarchy consists of two layers. At the lower layer, LCSs receive sensor signals and emit motor signals in parallel. At the upper layer, one LCS coordinates the LCSs of the lower layer by adjusting their excitation levels. The robot faces several tasks: following a light source (single goal), following a light source while avoiding a hot object (two conflicting goals), minimizing the average distance between two light sources (two simultaneous goals), or following a light source, reaching food when hungry and available, and avoiding predators (three mutually exclusive goals). A robot with one LCS at the lower layer can learn to follow the light source and that it can adapt to changes in the trajectory of the light source (different path or higher speed). With two LCS at the lower layer, one detecting light and one detecting hot objects, the robot either circumvents the hot objects or stops and waits until the light source reappears to follow it again. Here, the final action is the weighted sum of the actions that the lower-layer LCSs propose, with the excitation levels representing the respective weights. Similarly, with two LCS detecting the two different light sources, the robot can maintain the average distance to the two light sources. In the task involving three goals, the final action is just one of the actions that the lower-layer LCSs propose; the upper-layer LCS decides which action is executed, acting as a switch. With this architecture, the hierarchical LCS do not perform better than a single, monolithic LCS. This result is similar what is observed by Richter et al. (2008) (see Section 3.2.4). However, applying a two-phase reward policy, that is, first training and rewarding the lower-layer LCS independently before adding the upper-layer LCS, considerably improves the performance of the learning system.

Dorigo (1995) extends and applies the results of the simulated robot on on a real robot, the AutonoMouse (Figure 3.2). The author uses the simulated environment only to develop the learning architecture; learning happens from scratch on the real robot. Furthermore, the author uses the improved classifier system ICS, whose main difference to the original LCS are: calling the genetic algorithm when a steady state is reached instead of calling it every fixed number of cycles, a new mutation operator (called mutespec), and dynamically changing the number of active classifiers[2]. To study the parallel execution of several ICS, the authors develop ALECSYS, a tool which mainly addresses the speed problems in the execution of the several ICS by explicit high-level and low-level parallelization. The real robot learns the following behavior: playing with a light source (by following it) until it hears a predator, in which case it hides in its lair. They also show that the real robot can adapt to lesions, where the eyes are inverted, one eye is blind, or the motors are regulated incorrectly. While the results are convincing, they rely on a considerable amount of a priori knowledge of the problem, for example, the number of low-layer LCSs or how the upper-layer LCS coordinates the low-layer LCS (weighted sum or switch).

3.2.2 The event-based learning classifier system X-TCS

A similar study on multi-objective robot control has been carried out by Studley (2006). The author first investigates whether the zeroth-level learning classifier systems ZCS (Wilson, 1994) or the XCS can control the simulated Animat robot presented by Wilson (1987), such that the Animat can pursuit multiple objectives. Given a suitable parameterization, the ZCS can solve very simple problems, but fails at slightly more complex problems. The XCS, however, is able to solve the more complex problems and is less sensitive to a suitable parameterization. The author also evaluates X-TCS (Studley and Bull, 2005) on a real robot, the 'LinuxBot' (Winfield and Holland, 2000). The X-TCS is based on TCS (Hurst et al., 2002) and addresses the semi-Markov decision problem of assigning rewards to actions, the result of which take different amount of time. To address this issues, the X-TCS decouples the execution of the action from the reward mechanism. Instead, actions are supposed to be executed continuously (e.g., "drive forward") and the returning rewarding depends on how long it has taken to reach the goal. The LinuxBot is a platform with two motor-driven wheels and a trailing wheel. On the platform runs a miniature computer, which communicates via wireless Ethernet. The robot is designed to be able to run over long periods of time with a battery and electrical pickups that connect to the floor surface. In the experiments, the LinuxBot is equipped with three IR proximity sensors, two bumpers, and three light-dependent resistors, all facing to the front. With the X-TCS, the LinuxBot can optimally solve a two-objective problem with little need of parameter adjustment. To the contrary, the X-TCS appears to find its own appropriate level of discretization of the problem space. The results are encouraging but can only be considered as first results requiring a more thorough investigation. Furthermore, in the current form the X-TCS can only be applied to problems for which the goal or the optimal solution is known. It would be interesting to see how the X-TCS can be extended to problems where the goal is not known beforehand.

[2] In the original LCS proposed by Holland (1976), several classifiers may be active at the same time, unlike in later LCS versions.

3.2.3 XCS hardware implementation

Although the structure of the XCS lends itself to a hardware implementation, the only attempts towards a hardware implementation of an XCS, in fact any classifier system besides the LCT discussed in Section 2.1.5, has been carried out by Bolchini et al. (2005) with more details on a possible implementation on a field-programmable gate-array (FPGA) presented by Bolchini et al. (2006). To reach the goal of an implementation of the XCS on an FPGA, the authors first introduce $\mathrm{XCS_i}$, a version of the XCS that uses integer arithmetic instead of floating-point arithmetic. The authors derive two bounds on the number of integer bits to be used for integer arithmetic, so that the XCS can distinguish its classifiers as well as accurate from inaccurate classifiers. They also identify several necessary changes to avoid rounding errors and inaccuracies in the XCS algorithm: they reformulate the prediction update rule, set the minimum classifier fitness to 1 to avoid issues with null classifier fitness, and replace the calculation of the fitness value with a rough lookup table. A software version of $\mathrm{XCS_i}$ with 8, 10, and 16 bits solves the 11-, 20-, and 37-multiplexer problem. However, the learning rate decreases considerably with increasing problem size. The hardware implementation needs 150 bits of memory per classifier, which amounts to about 293 kB for the 2 000 classifiers of the 37-multiplexer problem. While the results of the software version of $\mathrm{XCS_i}$ are encouraging, the authors only present a design, not a final implementation. The final size of the FPGA implementation (e.g., required number of slices) is yet unknown (Bolchini, personal communication, 2006).

3.2.4 Distributed LCS

The population-based nature of the LCS and its genetic algorithm led to the thought of distributing the LCS and studying the effects of migrating classifiers among the distributed LCS instances, inspired by the migration component of Parallel Genetic Algorithms.

Bull et al. (2005) investigate whether rule migration improves the learning speed of the accuracy-based learning classifier system YCS. YCS is a simpler form of XCS, with the two main differences being that YCS has no triggered niche GA (and thus no mechanism to form maximally general classifiers) and that YCS does not execute any form of subsumption. The idea is to have a simpler version of XCS to study accuracy-based learning classifier systems. The authors use the 20-multiplexer problem as a benchmark. There are ten YCS instances which classify the same input data. A majority vote in the exploit step determines the overall classification outcome of the distributed LCS. With a given probability (0.1% or 1%), a YCS instance sends a given (1% or 10%) proportion of its population to another, randomly selected YCS instance. In every case, the number of steps needed to reach the optimum decreases by about 50% compared to a single YCS. Sending with the higher probability and sharing the higher proportion of the population decrease the average error. The improvements are not due to the larger overall number of classifiers, as a single YCS with the same number of classifiers does not show any positive effects. The number of steps needed to reach the optimum is further reduced by about 50% with niche-based rule-sharing, that is, the rules emigrate from the action set instead of the whole population.

Dam et al. (2005) investigate a client–server system consisting of LCS for distributed data mining, which Dam et al. (2008) refine and coin DLCS. The authors use UCS (Bernadó-Mansilla and Garrell-Guiu, 2003), an LCS that is similar to XCS but uses supervised learning instead of reinforcement learning, which is expected to be beneficial to classification tasks. Each client has its separate portion of the data on which it trains its classifiers. The clients periodically send their local models (i.e., their classifier populations) and training data that they could not classify correctly to the server. The server runs the received local models on the training data, gathers the output from the local models and fuses it. The authors investigate two fusion methods: knowledge probing, where the server trains its own LCS on top of the clients' LCS, and majority vote, where the majority of all client classifications determines the outcome. The authors investigate the effects of noisy data, client number and fitness-based rule sharing as done by Bull et al. (2005) on the DLCS. The investigations are performed with the 20-bit multiplexer. Concerning the fusion method, the results show that majority voting is more robust in noisy environments and thus preferred in their experiments. The learning rate increases with the number of clients, but the increase saturates after with six ore more clients. The learning rate also increases if rules are shared; however, in that case the overall population size increases. The results support the findings of Bull et al. (2005) that rule sharing is beneficial to learning in LCS. However, the benefits of adding an additional server instance remain unclear.

A similar system that employs the XCS is presented by Gershoff and Schulenburg (2007). However, each client and the server themselves consist of several XCS, called micro-agents, whose classification results are fused by majority voting. Furthermore, the server operates on the clients' classification results and not on the original data, similar to the knowledge probing technique described previously. In certain situations with the 6- and 11-multiplexer, the hierarchical system solves classification problems with far less data than a single-instance system.

Skinner et al. (2007) investigate the effect of adding migration pressure to the XCS by adopting migration strategies from multi-deme Parallel Genetic Algorithms. They set up eight XCS demes to classify noisy human electroencephalographic signals as being recorded while a participant performed the mental tasks "counting" or "figure rotation." Five additional parameters describe the migration policy. The migration rate specifies the proportion of an XCS deme population that participates in migration. The migration frequency specifies the number of exploration steps between two migrations. The hold-off period specifies a time period at the beginning of the experiments at which no migration occurs so that the XCS can evolve a sensible population. The classifier selection policy specifies whether the migrating classifiers are ranked by numerosity, by fitness, or randomly. And last, the classifier replacement policy specifies whether immigrating classifiers replace existing classifiers by fitness or randomly. Insertion and deletion is performed by the same algorithms as in the genetic algorithm. Fully connected XCS demes improve classification accuracy statistically significantly from about 78% without migration pressure to about 82% with migration pressure. Learning speed increases

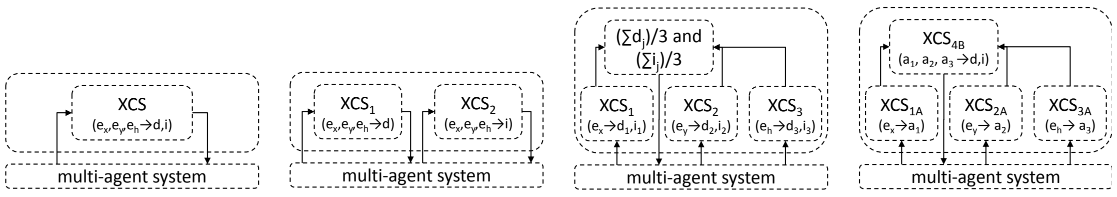

XCS
$(e_x,e_y,e_h \rightarrow d,i)$
multi-agent system
XCS_1
$(e_x,e_y,e_h \rightarrow d)$
XCS_2
$(e_x,e_y,e_h \rightarrow i)$
multi-agent system
$(\sum d_j)/3$ and $(\sum i_j)/3$
XCS_1
$(e_x \rightarrow d_1,i_1)$
XCS_2
$(e_y \rightarrow d_2,i_2)$
XCS_3
$(e_h \rightarrow d_3,i_3)$
multi-agent system
XCS_{4B}
$(a_1, a_2, a_3 \rightarrow d,i)$
XCS_{1A}
$(e_x \rightarrow a_1)$
XCS_{2A}
$(e_y \rightarrow a_2)$
XCS_{3A}
$(e_h \rightarrow a_3)$
multi-agent system

Figure 3.3: Single and distributed XCS architectures used by Richter et al. (2008). From left to right: single XCS; two XCSs for each output dimension; three XCS for each input dimension, combined by averaging; four XCS, three for each input dimension and one for combination. From Richter et al. (2008).

significantly, too. Bi- and unidirectionally connected XCS demes also show improved classification accuracy but sometimes show lower learning speeds. Migration pressure reduces the final size of the classifier population to about 40% of the classifier population of the single XCS. While the improvements are statistically significant, the absolute increase in classification accuracy is rather small.

Richter et al. (2008) investigate the learning rate of three different distributed XCS systems in a simulated chicken-coop problem: densely packed chicken in a chicken coop tend to chase, pick and ultimately kill wounded chicken, which can be prevented by dispersing the chicken clusters with a sound signal of some duration and intensity. The input values of the problem are the combined (x, y) position of the chickens as well as their overall heading; the output values are the intensity and duration of the sound signal. Three different distributed systems are analyzed(see Figure 3.3). In the first system, two XCS operate in parallel and determine the duration and intensity of the sound signal independently. In the second system, three XCS operate in parallel, where each XCS uses one of the input values to determine both output values; these are then averaged to produce the final values of the duration and intensity of the sound signal. In the third, hierarchical system, three XCS operate in parallel, where each XCS uses one of the input values to determine whether a noise intervention is preferred; a fourth XCS maps the outputs of the three XCS to the duration and intensity of the sound signal, if any. Unlike Dam et al. (2008), the hierarchical XCS system shows no advantages. The second system shows the best learning rate.

Although the authors did not investigate the following aspect further, their analysis of distributed XCS also indicates how to separate the problem space. While other studies separate the problem space only by dividing it into several clusters, in this study the problem space is separated along the input and output dimensions: in the first system along the output dimensions, in the second system along the input dimensions, and in the third system along the input and output dimensions. The results may suggest that separation along the output dimensions is detrimental, while separation along the input dimensions may be beneficial to the learning speed.

In all presented works, the distributed XCS instances solve one common classification problem. In this thesis, however, the distributed XCS instances solve (slightly) different problems.

3.3 Dynamically reducing power consumption

One of the most elaborated fields of specialized adaptation at chip level are dynamic adaptations to reduce power consumption. The following will give an overview of these common, specialized adaptation circuits and techniques. As outlined in Section 2.3.1, power consumption depends on several parameters, of which only voltage, frequency, and switching activity can be altered dynamically at run time. As power consumption decreases quadratically with voltage reduction, dynamically reducing the voltage of a chip is a prime goal to reduce the chip's power consumption. Power consumption also decreases linearly with frequency reduction, so, if idle periods exist or maximum performance is not paramount, reducing the clock speed of a chip is also of interest.

Adjusting the voltage and frequency of a chip at run-time is known as dynamic voltage and frequency scaling (DVFS) and is one of the most prominent forms of dynamic adaptation to reduce power consumption. For example, Gutnik and Chandrakasan (1997) present a circuit to add variable voltage levels and frequencies to a digital signal processor (DSP). The authors report power savings of 30%–50% in the typical case and an order of magnitude under certain circumstances. They use the fill level of an input FIFO buffer to estimate the sufficient processing rate and the according voltage level. Kawaguchi et al. (2002) use off-the-shelf processors to realize a power-saving movie-playback system (MPEG4). Unlike Gutnik and Chandrakasan (1997), the controller is realized as a software component. With only two voltage levels, the authors' system saves up to 88% in power consumption compared to a system that idles if there is no work and up to 63% compared to a system that is put into sleep state when possible, while maintaining the real-time properties of the movie playback.

Other techniques of dynamic adaptation to reduce power consumption address the third power-consumption parameter that can be altered dynamically at run time, the switching activity. A transistor switches and draws current if its input toggles, which can be the clock or regular input, and if it is connected to the power supply. This directly translates to the further techniques to dynamically reduce power consumption: clock gating (which turns off the clock switches), operand isolation (which turns off the input switches), and power gating (which disconnects the transistor from the power supply).

In clock gating, the clock of a register is disabled by additional combinational logic to prevent unnecessary switching when the register's contents are not altered. Benini et al. (1999) analyze the implemented Boolean functions to identify these situations and stop the clock of a register. Applied to a standard industry benchmark, this reduces power by up to 34% with an average increase of 11% in the number of gates and 8% in delay. Hurst (2008) present an automated process that is scalable to large designs. It operates on the netlist level and uses the signals of the already existing circuit to control the additional logic of the clock gating mechanism to save area. Applied to an industry-supplied benchmark, the process reduces power by 14% on average.

In operand isolation (Correale, 1995), the inputs of a section of circuitry are disabled by additional combinational logic when the result of the circuitry is not of interest (e.g., unused execution units in an arithmetic logic unit (ALU) of a processor). Tiwari et al. (1995) present operand isolation at the gate level. Kapadia et al. (1999) present operand isolation on data-path buses that gate the enable signal of registers instead of directly blocking the input signals. Münch et al. (2000) present a technique that operates at the register-transfer level.

While clock gating and operand isolation help to reduce dynamic power dissipation, power gating also reduces static power dissipation, as it disconnects sections of the circuitry from the power supply. This is typically done under software control (mostly the operating system) when it is (supposedly) known that some sections of the circuitry will not be used for some time. To disconnect the circuitry from the power supply, a so-called sleep transistor or power switch is inserted between the actual ground and the circuit ground (called virtual ground) or between the actual supply voltage and the circuit supply voltage (Mutoh et al., 1995). Power gating is predominantly realized with multi-threshold complementary metal-oxide-semiconductors (MTCMOS). When the sleep transistor is turned off, the virtual ground charges up close to the supply voltage and no leakage occurs anymore. However, when the sleep transistor is turned on again, the virtual ground has to discharge through the sleep transistor. Thus, power gating incurs large wake-up latencies and power penalties that have to be traded-off with the power savings. This is in contrast to clock gating and operand isolation, which can be turned on and off at clock speed and where the additional power consumption is usually negligible. Multiple sleep modes with increasing latencies and power savings help to find optimal trade-off points for different time periods of circuit inactivity (Agarwal et al., 2006). Alternatively, if neighboring circuits show opposite patterns of power gating, the power penalty can be reduced by charge recycling (Schweizer et al., 2010).

The presented techniques certainly help to dynamically reduce power consumption, but they are specialized techniques to solve one particular problem, namely reducing power consumption. In parallel, but independently, techniques have been developed to increase fault tolerance through redundancy. Only recently, the implications of increasing both fault tolerance and power savings have been studied (Melhem et al., 2004; Zhang and Chakrabarty, 2003), requiring several careful design consideration (Degalahal et al., 2005; Zhu and Aydin, 2009). The proposed design methodology for self-adaptation at chip level allows to combine the goals of fault tolerance and power saving as well as other goals instantly with an appropriate reward function in a "good enough" manner. This allows to meet the short time to market and develop more elaborate and specialized techniques afterwards in a future revision.

3.4 Summary

The review of related work shows that, while there is work on self-adaptation, mainly within the Autonomic and Organic Computing initiatives, no work considers self-adaptation at the basis of virtually all computing, the chip. Adaptive and adjustable circuits are numerous, but none is self-adaptive, that is, no circuit adapts itself; instead, it (only) adapts some other part of the system. As outlined in the introduction, this inflexibility reduces design reuse and hinders technological advancement.

Machine algorithms that are implemented in hardware come closest to the proposed self-adaptive controller, which is also based on a machine learning algorithm. However, the suggested hardware implementations are (too) large and are usually intended to speed up machine learning, not to adapt a SoC. While the Aron system realizes some form of self-adaptation, it is only applicable to regularization problems and it must be initialized by an expert of the system under control, who not only must know the optimal system states but also how to reach the optimal system state from other states.

So far, LCSs that control hardware are realized as software components; no work has considered using hardware-based LCSs. Additionally, the LCSs control robots and similar devices but not chips. Further, the LCSs do not come with a design methodology or, more generally, the LCSs do not learn the classifiers anywhere else beforehand but only just in time on the controlled device. The proposed hardware implementation of XCS on an FPGA is still in its design phase and has not yet been finished. Besides, its hardware requirements will probably be large.

All setups that distribute learning classifier systems across many learning nodes solve one common problem. No work considers learning nodes that solve (slightly) different problems and increase their learning capabilities by cooperation through classifier exchange. A variation that comes closest to the work presented in this thesis is presented by Richter et al. (2006), who separate the input space, train LCSs on each dimension and join the results in a higher-level LCS. Still, the LCSs solve one single problem.

In summary, self-adaptation at chip level has not been investigated so far. Existing techniques that could be used to realize self-adaptation at chip level, such as machine learning algorithms implemented in hardware or sophisticated lookup tables, are not ready for chip-level control. Further, distributed LCS have only been used to solve one common problem.

4 Self-adaptation at chip level using the learning classifier systems XCS

This chapter presents the main contributions of this thesis. It starts with Section 4.1 that presents the proposed design methodology for self-adaptation at the chip level. The two important aspects of the design methodology are a machine learning algorithm and two different versions of the self-adaptive controller at design time and run time. The chapter continues with Section 4.2 that describes the methods for self-adaptation using cooperation. Section 4.3 presents the operator-allocation problem, an application example, which is also used as a benchmark. Section 4.4 extends the current XCS theory to cover complex problems that are encountered with self-adaptation at chip level. Section 4.1.2 presents the method of subsuming after learning, which is well-suited for the presented two-phase methodology. The chapter ends with Section 4.5, which gives an overview of the simulation library that is used in the experiments to co-simulate the hardware and software of the SoCs and to train and simulate the XCS.

4.1 Design methodology

The proposed design methodology builds on the Autonomic System-on-Chip (ASoC) project presented in Section 2.2. An ASoC extends a regular SoC by an autonomic layer, whose autonomic elements monitor and control their functional counterparts in the functional layer (see Figure 2.5a). Each autonomic element contains a monitor, an evaluator, and an actuator, as well as a communication interface.

The proposed design methodology is concentrated on the evaluator of the Autonomic System-on-Chip: what kind of evaluator is suitable to fulfill the goals of the ASoC project and how does the design of this new kind of SoC component look like? The other parts of an autonomic element are already investigated in the ASoC project. The proposed design methodology corresponds to the well-established Y-shaped design flow (see Figure 2.5b). After the requirements of the SoC application are mapped to the characteristics of the architecture, an optimization cycle begins that looks for optimal trade-offs between performance, power, and reliability of the chip. The proposed design methodology contributes methods to optimally select the parameters of the evaluator (see Section 4.4) and a simulation model to evaluate the resulting design including the evaluator, the proposed self-adaptive controller (see Section 4.5).

This section presents the two important aspects of the proposed design methodology: first, a machine learning algorithm to reduce the design effort and to realize self-adaptation at run time and, second, two different versions of the self-adaptive controller at design time and run time.

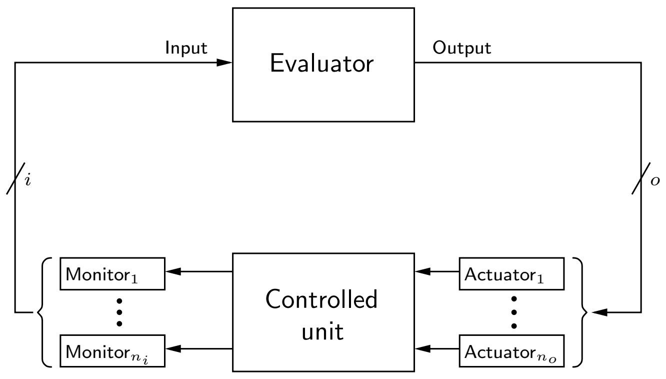


Figure 4.1: Self-adaptation system. n_i monitors sense the state of the controlled unit and serve as input to the evaluator. The evaluator's output signal is wired to n_o actuators, which change the state of the controlled unit, ideally to a better operating point.

4.1.1 Machine learning algorithm

As the self-adaptation system is supposed to not only execute appropriate actions under different operating conditions, but to also adapt itself to hardly known or even unknown variabilities, a machine learning algorithm is required. This section determines the requirements of a self-adaptation system and its machine learning algorithm, which will turn out to be the reinforcement algorithm XCS (Wilson, 1998).

Naturally, a self-adaptation system will provide the usual input and output interface of any other adaptive circuit (see Figure 4.1): 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.[1] To allow reusability, the monitor and actuator signals will be actual bit vectors at the input and output of the self-adaptation system. They respectively consist of n_i and n_o individual monitors and actuators, providing i and o total bits of monitor and actuator data. At the heart of the self-adaptation system lies the *evaluator*, which reasons about the current chip state to suggest a preferably optimal action that moves the controlled unit to the desired chip state.

To allow reusability, the evaluator has to be as ignorant as possible to the meaning of the input and output signals. With an ignorant evaluator any kind of monitor or actuator can be used that communicates its current state and accepts commands as a digital bit signal. This includes the majority of the existing monitors (e.g., Nicholaidis flip-flop (Nicolaidis, 1999) or RAZOR element (Ernst et al., 2003)) and actuators (e.g., current DVFS components, see Section 3.3).

In current special-purpose adaptive circuits, 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, this thesis proposes to use an evaluator that automatically learns the optimal actions for a given situation by itself. Of course, as learning involves

[1] In the literature on learning classifier systems, monitor and actuator are called sensor and effector, respectively. The terms are used interchangeably in this thesis.

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 systems that use special-purpose adaptive circuits. From the machine learning algorithms, the reinforcement algorithms seem to be 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 (Sutton, 1988) 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 the goal of a self-adaptation method. Additionally, if the designer knows the best action for a given system state, it is desirable that he or she could tell the 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 (Holland, 1976) 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.

There exist numerous machine learning algorithms that could have been used for the evaluator, but only a few provide efficient hardware implementations, most notably neural networks (Dias et al., 2004; Widrow et al., 1994) and, more recently, support vector machines (Anguita et al., 1998, 2003; Irick et al., 2008). However, the knowledge that is learned by these algorithms is opaque. Additionally, their hardware implementations are large (see Section 3.1.3 for details). Further, hardware realizations of other machine learning algorithms such as swarm intelligence (Bonabeau et al., 1999; Tarasewich and McMullen, 2002) or chemistry computing (Dittrich, 2005) are expected to have a large hardware overhead and a high power consumption, as the minimum number of swarm members or chemical elements that the algorithms need is large (i.e., their "critical mass" is large). On the other hand, learning classifier systems (LCS) are able to generalize over several similar cases using wild cards, leading to fewer necessary classifiers. Furthermore, Wilson (2002) and Dixon et al. (2003) present algorithms that reduce the number of rules while maintaining the performance of the LCS, promising the possibility of a smaller and more effective implementation than other schemes. The learning classifier system XCS (Wilson, 1998) is reported to learn accurately and completely (Kovacs, 1997). XCS has also been shown to be implementable in hardware so that it allows chip-level control. Bolchini et al. (2006) present the design of a possible implementation of the XCS on an FPGA. Zeppenfeld et al. (2008) show an efficient hardware implementation, which also retains the self-adaptation capabilities of the XCS. Their implementation has a period of 7.6 ns.

For the reasons given in the preceding paragraphs, this thesis proposes to use the XCS as the evaluator of a self-adaptive controller. If the designer can formulate the cost function of a chip state such that it can be implemented in hardware, it can be used as a reward function to realize the self-adaptive controller.

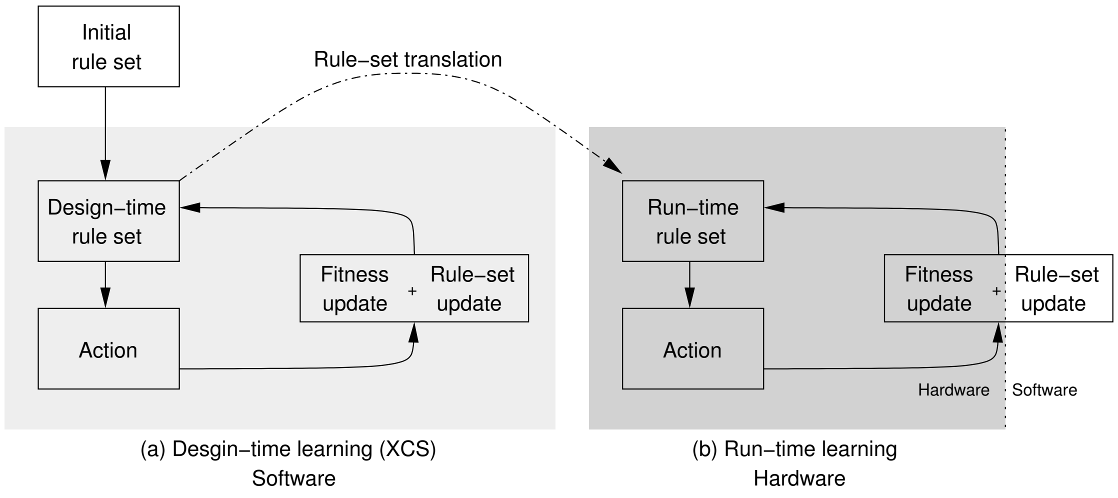


Figure 4.2: Learning at design and run time. (a) Learning at design time with a full-fledge XCS. (b) Learning at run time with a compatible LCS optimized for a hardware implementation. The initial run-time rule set is translated from the resulting design-time rule set. At run time, only parts of the learning aspects might be present to realize a lightweight hardware implementation.

4.1.2 Design-time and run-time learning

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. This problem is addressed by the proposed design methodology by separating the design-time from the run-time implementation with the following two-stage approach (compare to Figure 4.2):

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

2. Before run time, the design-time rule set gets translated into a form that is suitable for the hardware implementation (*rule-set translation*). At run time, a hardware-based learning classifier system uses the resulting rule set (*run-time rule set*) to continue learning and selecting (preferably optimal) actions.

The idea is that the XCS learns a rule set that allows the hardware implementation to adapt to the actual manifestation and conditions of a particular chip and even to unexpected events, despite its limited learning capabilities. With this setup, all design-time resources are available for a capable software implementation (the XCS) and the acquired design-time knowledge can be transferred to a lightweight hardware implementation. This thesis uses the LCT as a hardware implementation of a learning classifier system, which has been developed by Zeppenfeld et al. (2008).

The LCT has no component that generates rules or even initial rules. For this, this thesis proposes the following algorithm to translate the design-time rules into a form suitable for the LCT, which ensures that the XCS and the LCT classifiers match the same input values.

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

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, this thesis proposes to use 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.

As the chip area that is necessary to store the classifiers in memory constitutes the largest part of the hardware implementation, it is desirable to minimize the number of necessary classifiers to keep the chip-area requirement small. Therefore, when analyzing the run-time aspects of the proposed design methodology in the experiments of this thesis, translating both all XCS rules to corresponding hardware rules (all-XCS translation) and only the top performing rules (top-XCS translation) are considered.

After design-time learning, all the important, accurate, and maximally general classifiers do exist. However, there still will be inaccurate and too specific classifiers since the GA has evolved new classifiers until the end of the learning phase. These classifiers are not necessary for a successful run-time self-adaptation system. The following introduces an algorithm which selects the optimal classifiers from the classifiers resulting from design time. The method allows reducing the population size considerably and works similarly to the methods GA subsumption and action set subsumption, which are described in Section 2.1.2. Additionally, the already mentioned algorithms to reduce the number of rules while maintaining the performance of the LCS (Dixon et al., 2003; Wilson, 2002) could be applied, although this is not investigated in this thesis.

The algorithm consists of two steps:

1. Subsumption: Try to subsume every pair of classifiers. Let cl_1 and cl_2 be two classifiers with the same action. The classifier cl_1 subsumes cl_2 if the following conditions are met:
    (a) The condition of cl_1 covers the condition of cl_2.
    (b) cl_1 is at least as accurate as cl_2.
    (c) cl_1 is experienced enough.
2. Deletion of superfluous classifiers: A classifier is not required and thus deleted if
    (a) other classifiers also cover its condition and
    (b) it is not accurate or not experienced enough.

GA subsumption and action set subsumption, if applied during learning, may cause an extended learning time or even poor performance, as they allow over-general, short-time accurate classifiers to subsume accurate classifiers (Butz et al., 2003) and thus hinder the evolution of optimal classifiers. As the presented subsumption algorithm is applied after learning, these disruptive effects cannot occur. For online learning scenarios, the standard subsuming method might be more adequate, since the learning process is ever continuing.

4.2 Self-adaptation using cooperation

The proposed design methodology also covers MPSoCs by using cooperating self-adaptive controllers, which scale with the number of cores of the MPSoC. Cooperation is achieved by exchanging knowledge between the self-adaptive controllers of similar units on a SoC such as processing cores. The cores do not have to be identical, though; for example, the cores may have different cooling properties because of their physical placement on the chip. The effects of exchanging classifiers among cores with different cooling properties is analyzed in an application example that is based on the Cell processor.

To simulate distributed self-adaptation systems, the XCS implementation has been extended by a migration component, a topology module and an additional classifier deletion round, which are inspired by the work of Skinner et al. (2007). Furthermore, a remote temperature module, which maintains a global temperature distribution, is added to the simulation library, which is described at the end of this chapter in Section 4.5. The migration component allows distributed XCS instances to send and receive classifiers. The topology module sets up the topology of the directed classifier paths between the XCS instances. The additional deletion round deletes excessive classifiers caused by immigration. The following subsections detail the three extensions.

4.2.1 Migration strategies

The *migration component* is active in every explore step of the XCS instance. First, it selects a predefined proportion of classifiers as emigrants according to a predefined emigration strategy, and sends the emigrants to the *target peers* of the XCS instance. Then, it receives a given number of immigrants from the *source peers* of the XCS instance, adds the immigrants to the local classifier population, and, if the maximum population size N is exceeded, deletes classifiers according to a predefined deletion strategy.

Currently, the migration component supports the following migration strategies:

- The *random* strategy (denoted by R) selects the emigrating classifiers at random. This allows a uniform distribution of all classifiers, but has the risk that unhelpful or harmful classifiers spread through the XCS populations. Dam et al. (2005) reports good results when emigrating classifiers at random.
- The *numerosity* strategy (denoted by N) selects the classifiers according to their numerosity, highest numerosity first. This strategy is expected to make numerous local classifiers accessible to other XCS instances. However, in static populations, the same classifiers will be copied repeatedly, so that the target peers do not receive new classifiers.

- The *fitness* strategy (denoted by F) selects the classifiers according to their fitness, highest fitness first. The expectation is that a fit classifier may benefit the neighboring XCS instance, too, possibly after a small modification. However, as some properties of neighboring cores may differ, the immigrating classifiers may hamper the evolution of the present classifiers.

The migration component adds three additional parameters to the simulation:

- $\delta_{\text{emigrants}}$ defines the proportion of the classifiers that the XCS selects as emigrants.
- δ_{hold} defines the initial delay during which no migration occurs (so that the initial learning takes place without migration and no purely random classifiers are distributed).
- $\mathcal{M}$ identifies the selected migration strategy (R, N, or F, as mentioned previously).

Typical values are $\delta_{\text{emigrants}} = 0.5\%$ and $\delta_{\text{hold}} = 0.001 N_{\text{steps}}$.

4.2.2 Topologies

The *topology module* defines a directed graph between XCS instances that send their classifiers. With an according configuration file, the module can realize any topology. Figure 4.3 shows some typical topologies:

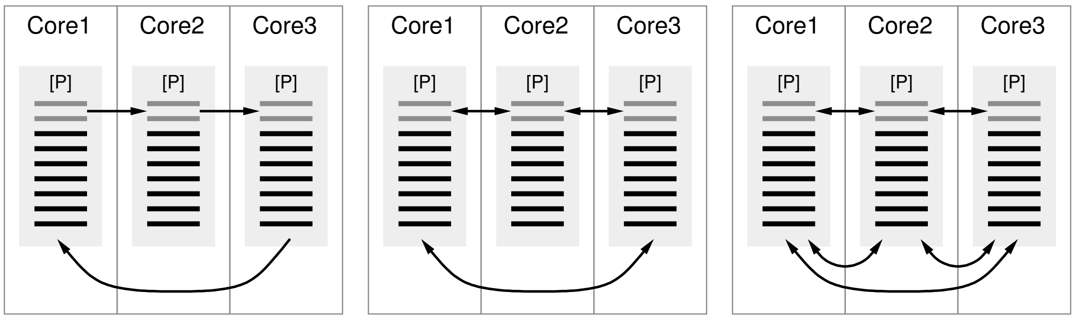
- In the *unidirectional ring* (denoted by U), every XCS instance receives immigrants from its left neighbor and sends emigrants to its right neighbor.
- In the *bidirectional ring* (denoted by B), every XCS instance receives immigrants from and sends emigrants to both its left and right neighbors.
- In the *complete graph* (denoted by C), every XCS instance receives immigrants from and sends emigrants to every other XCS instance.

The parameter $\mathcal{T}$ indicates the selected topology: U for the unidirectional ring, B for the bidirectional ring, and C for the complete graph.

4.2.3 Deletion strategies

Adding the immigrating classifiers to the population may exceed the maximum population size. In that case, the classifiers are deleted by the migration component in an additional deletion round according to strategies similar to the deletion strategies of regular XCS operation:

- When *deleting by fitness* (denoted by F), the deletion probability is inversely proportional to the classifier's share of the average fitness of the population. Classifier with low fitness are deleted preferably.



(a) Unidirectional ring (b) Bidirectional ring (c) Complete graph

Figure 4.3: Typical topologies of communicating XCS instances.

- When *deleting by action set size* (denoted by A), the deletion probability is proportional to the average size of the action sets that contained the classifier. Classifiers that are often in large action sets (where all classifiers propose the same action under the same condition) are deleted preferably.
- When *deleting by prediction error* (denoted by E), the deletion probability is proportional to the classifier's share of the average prediction error of the population. Classifiers that are inaccurate are deleted preferably.[2]

The parameter $\mathcal{D}$ identifies the deletion strategy: F for deletion by fitness, A for deletion by action set size, and E for deletion by prediction error.

4.3 Formalization of the operator-allocation problem

"The key to the reliability problem might be to exploit the abundance of transistors—use Moore's law to advantage. [...] [A] shift toward parallelism to deliver higher performance is in order, and thus *multi* might be the solution at all levels—from multiplicity of functional blocks in a design to multiple processor cores in a system" (Borkar, 2005, p. 15, emphasis in original). In the *multi*-world it is favorable to allocate several operators for the same operation to overcome reliability issues with single operators; the chip applications can profit instantly from a hardware-based solution without the need to adjust software.

In this spirit, this thesis analyzes the application of the self-adaptive controller to the problem of dynamically allocating several operators for a single operation to increase the overall reliability by spatial redundancy. The *operator-allocation problem*, which is defined later, also serves as a benchmarks for chip-level self-adaptation, as no such

[2] The original C-implementation of the XCS contained a bug when deleting by prediction error: in the rare case that the prediction error of the classifiers was zero, a division by zero error occurred during the calculation of the deletion probability. This has been fixed by deleting at random in that case. I thank Gunnar Arndt for discovering this bug and submitting an adequate fix.

benchmark is known. Unlike control problems, the operator-allocation problem allows to evaluate the basic performance, the self-adaptation capabilities and the costs of the proposed design methodology in a systematic way.

The problem definition is guided by the following reasoning. Concerning the basic performance, the problem is simple enough so that the optimal solution is known, 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. To determine the self-adaptation capabilities, the operator-allocation problem allows the simulation of unforeseen events. Lastly, the operator-allocation problem also allows to evaluate the generalization capabilities of the proposed system, a capability that is helpful to reduce design efforts. No other problem definition is known that shows these properties.

The (c, t) *operator-allocation problem* is defined as follows: given c operators, select $t \leq c$ operators, which are to be allocated for an operation, by indicating one of the $\binom{c}{t}$ possible allocations, while some of the c operators are known to already execute some other operation and are thus not available. The input interface of the self-adaptive controller (coming from the monitors) consists of c signals that indicate the free and occupied operators. The output interface of the self-adaptive controller (going to the actuator) consists of

$$n = \left\lceil \log_2 \left(\binom{c}{t} + 1 \right) \right\rceil \tag{4.1}$$

actions that indicate either one of the $\binom{c}{t}$ possible allocations or that there is no valid allocation (e.g., when all operators are occupied). The action "no valid allocation is possible" is encoded as 0 (*action zero*), all other actions are encoded with the combinadic function (McCaffrey, 2005). An allocation is *valid* or *correct*, if it allocates only free operators. The problem can easily be made more difficult by imposing additional constraints on the operators that are to be allocated; for example, the operators can be required to maximize the combined reliability of the selected operators, to have a maximum distance to already occupied operators so that hot spots are avoided, or a combination thereof. This thesis sticks to the basic version for the first evaluation of a self-adaptation system and notes that additional constraints can be easily added by altering the reward function, which the designer is supposed to provide. Also, the thesis does not further consider the comparator that comes after the operators and that is necessary to detect and correct errors (e.g., by majority voting). The operator-allocation problem is used as an additional application example and benchmark in the experiments of this thesis.

The unexpected event for the operator-allocation problem is the total failure of one or more operators. A failed operator is monitored as free, but an allocation of this operator fails (e.g., indicated by a missing "success" signal), that is, although the operator looks available to the XCS, it cannot be allocated. This might be a typical symptom if the engineer did not consider the failure of an operator at design time and did not instruct the monitors to report failed operators as not available. It is assumed that a failed operator never recovers.

From a self-adaptation point-of-view, the operator-allocation problem is not trivial.

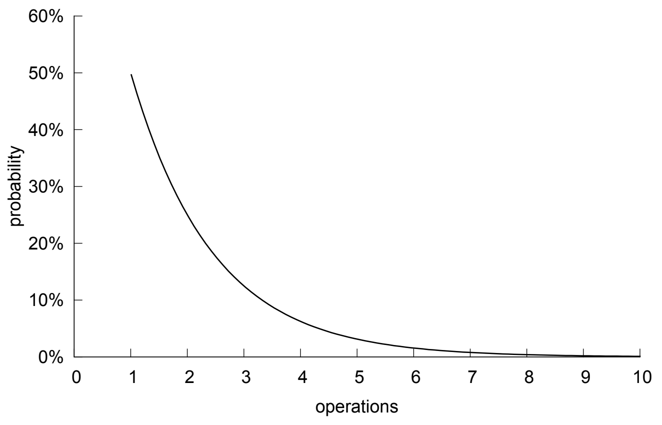


Figure 4.4: Probability that a random allocation is valid.

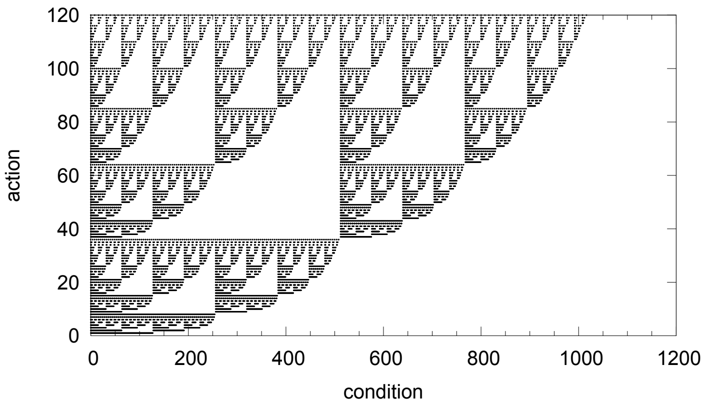


Figure 4.5: Valid allocations in the $(10, 3)$ operator-allocation problem. The dots are enlarged for visibility; actually, they cover only about 12% of the chart area, corresponding to the probability that a random allocation of three operators is valid.

As can be seen in Figure 4.4, 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 the number of operations t to allocate. This probability is independent of the total number of operators c. The probability that a random allocation is valid lies at about 12% for allocating three operators and at about only 3% for allocating five operators.

Furthermore, the valid allocations are distributed unevenly, as depicted in Figure 4.5, which shows the valid allocations in the $(10, 3)$ operator-allocation problem. The x axis shows the $2^{10} = 1024$ possible occupations of the ten operators. The y axis shows the $\binom{10}{3} = 120$ possible operator-allocations as actions. The diagram contains a dot for every valid action under the given occupation condition. For extended versions of the problem, the reward could scale with the goodness of the action, for example with the combined reliability of the allocated operators.

4.4 Extended XCS theory

At the heart of the current theory on XCS lies the problem dimension k, which is needed to find the minimum value for the maximum population size N of the XCS as well as the corresponding don't-care probability $P_{\#}$. In the work of Butz et al. (2003) and Butz et al. (2004b) on XCS theory, which is presented in Section 2, specifying k was simple in so far as the classifiers for all problem instances need the same number of specified bits. Furthermore, the classifiers are not overlapping. For example, to correctly solve any problem instance of the 6-multiplexer problem, the XCS only needs to know the value of 3 bits: the value of the two indexing bits and the value of the indexed bit (e.g., $m_6(\underline{01}11\underline{0}1) = 0$). Problems like the multiplexer problem, where the number of relevant bits to solve a problem instance is constant across all problem instances, are called *regular* problems in this thesis. The current XCS theory is not sufficient for the problems considered in this work, as it considers problems where the number of relevant bits differ among the problem instances.

For example, in the $(4, 2)$ operator-allocation problem, the number of relevant bits depends on whether an allocation is possible or not. If an allocation is possible (e.g., given the input state 1100), the number of relevant bits equals to the number of operations to allocate, as the XCS only has to know that the specified operators are not occupied (i.e., have the value 0). If an allocation is not possible (e.g., given the input state 1110), the number of relevant bits equals the minimum number of operators where no allocation is possible, as the XCS has to know that at least so many operators are occupied (to return action zero). The effect of different number of relevant bits is more prominent if the number of available and to-be-allocated operators differ more. Problems like the operator-allocation problem, where the number of relevant bits is not constant across all problem instances, are called *complex* problems in this thesis.

As the current XCS theory depends on knowing the problem dimension k of the problem that is to be analyzed but that problem dimension is not known for complex problems, the current theory cannot be applied readily to complex problems. This section shows how to estimate an equivalent problem dimension k for complex problems including problems with classifiers that overlap or have different individual problem dimensions. The equivalent problem dimension makes current XCS theory accessible for complex problems. The following approach is analogous to the existing XCS theory: first, analyze the conditions under which optimal classifiers can exist (to answer the question which conditions must be met so that optimal classifiers can evolve) and then, assume that, if these conditions are met, the optimal classifiers will evolve.

The population size N and the number of required learning steps grow with the size of k (Butz et al., 2004a). Thus, k should be as small as possible but still large enough to ensure that all optimal classifiers can evolve. If the requirement that k should be as small as possible could be neglected, k could simply be set to the maximum $k_{\max}$ of all occurring classifier specificities $k_{cl} = \sigma(cl)$:

$$k_{\max} = \max_{cl \in [P]} (k_{cl}) \tag{4.2}$$

As the classifiers with the most specified bits are the most difficult to evolve, $k_{\max}$ ensures that all optimal classifiers can evolve.

Although $k_{\max}$ ensures that all optimal classifiers can evolve, it may also lead to a very large classifier population only because a single niche has a very high problem order. To reduce the necessary amount of memory to store the classifiers on the chip, it is desirable to reduce the classifier population size, hence increasing the evolutionary pressure by using a $k < k_{\max}$ that still allows the evolution of optimal classifiers. A smaller k comes with the risk that not all optimal classifiers can evolve. It is therefore necessary to decide which classifiers contribute the most towards solving the given problem. This is realized by assigning a weight w_{cl} to each classifier according to its importance. The problem dimension k is then calculated as the weighted average of the classifiers' problem dimensions:

$$k_{\text{avg}} = \sum_{cl \in [P]} \frac{w_{cl}}{w} k_{cl} \tag{4.3}$$

Here, k_{cl} is the problem dimension of a classifier (i.e., the number of bits that are specified in its condition), w_{cl} is the individual classifier weights, and

$$w = \sum_{cl \in [P]} w_{cl} \tag{4.4}$$

is the total weight.

The weight function that represents the importance of an individual classifier is motivated by the following insights:

1. A classifier that matches frequently is more important than a classifier that matches only infrequently, as the former contributes more to the overall performance of the XCS than the latter. For example, consider the aforementioned problem where niches with problem dimension $k_{\max}$ occur very rarely. If the classifiers that represent these niches are not present in the classifier population, the XCS will still choose the best action in most cases and only in some rare cases it will choose a poor action. Allowing this small performance degradation allows smaller k values and thus smaller population sizes. The weight of a classifier is thus proportional to its probability of being part of the action set.

2. When the same action is propagated by many classifiers, it is not important that all of the classifiers exist. For example, consider a niche that is matched by many classifiers that propose the same action with the same reward prediction. If one of these classifiers is missing, the overall performance of the XCS will not degrade concerning this niche. The weight of a classifiers is thus inversely related to the number of classifiers in the action set.

From these insights, the definition of the classifier weights w_{cl} follows as:

$$w_{cl} = \Pr(cl \in [A]) \cdot \frac{1}{\|[A]\|} \cdot k_{cl} \tag{4.5}$$

Of course, $[A]$ depends on the classifier population $[P]$, the current input state s and the selected action a. As the classifier population is assumed to be optimal, it is fixed. Let a_o be the action selected in explore mode and a_i the action selected in exploit mode. Further, let $[A]_s^a$ denote the action set resulting from the input state s and the selected action a. Then, introducing the input state s explicitly in Equation 4.5 leads to

$$\begin{aligned} w_{cl} = \sum_s &\Pr(cl \in [A] \mid s, \text{explore}) \cdot \Pr(s) \cdot \Pr(\text{explore}) \cdot \frac{1}{\|[A]_s^{a_o}\|} \cdot k_{cl} \\ &+ \Pr(cl \in [A] \mid s, \text{exploit}) \cdot \Pr(s) \cdot \Pr(\text{exploit}) \cdot \frac{1}{\|[A]_s^{a_i}\|} \cdot k_{cl} \end{aligned} \tag{4.6}$$

where s runs over all possible input states. Pr(exploit) is given as the XCS parameter P_{explt} and $\Pr(\text{explore}) = P_{\text{explr}} = 1 - P_{\text{explt}}$. The variables k_{cl}, P_{explt}, P_{explr} do not depend on the input state. Usually, the input states are uniformly distributed, hence $\Pr(s)$ is constant. With the definitions

$$p_{\text{explore}}(s, cl) = \Pr(cl \in [A] \mid s, \text{explore}) \cdot \frac{1}{\|[A]_s^{a_o}\|} \tag{4.7}$$

and

$$p_{\text{exploit}}(s, cl) = \Pr(cl \in [A] \mid s, \text{exploit}) \cdot \frac{1}{\|[A]_s^{a_i}\|} \tag{4.8}$$

Equation 4.6 can thus be rewritten as

$$\begin{aligned} w_{cl} &= k_{cl} \cdot \Pr(s) \cdot \left(\sum_s \Pr(cl \in [A] \mid s, \text{explore}) \cdot P_{\text{explr}} \cdot \frac{1}{\|[A]_s^{a_o}\|} \right. \\ &\qquad \left. + \sum_s \Pr(cl \in [A] \mid s, \text{exploit}) \cdot P_{\text{explt}} \cdot \frac{1}{\|[A]_s^{a_i}\|} \right) \\ &= k_{cl} \cdot \Pr(s) \cdot \left(P_{\text{explr}} \cdot \sum_s p_{\text{explore}}(s, cl) + P_{\text{explt}} \cdot \sum_s p_{\text{exploit}}(s, cl) \right) \end{aligned} \tag{4.9}$$

The default action-selection method for the explore mode is random selection, thus

$$\Pr(cl \in [A] \mid s, \text{explore}) = \begin{cases} \frac{1}{n} & \text{if } cl \text{ matches } s \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

The default action-selection method for the exploit mode is winner-takes-all selection. If the matching classifiers have the same fitness, the classifier that predicts the highest reward gets chosen. When there are several classifiers that predict the highest reward, it is safe to assume that, during learning, each of these classifiers gets selected equally likely. Let $n_a(s)$ be the number of actions that are proposed by the matching classifiers that predict the highest reward. Then,

$$\Pr(cl \in [A] \mid s, \text{exploit}) = \begin{cases} \frac{1}{n_a(s)} & \text{if } cl \text{ matches } s \text{ and } cl.r \text{ is maximal} \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

Knowing the equivalent problem dimension k_{avg}, N can be approximated with the Equations 2.14, 2.16, and 2.19.

The presented formalization assumes that the environmental niches are known, so that k_{cl} is known and p_{explore} and p_{exploit} can be calculated. This is the same constraint that applies to the current XCS theory. Of course, in real-world problems the optimal classifiers are not known beforehand, and thus k_{avg} cannot be calculated directly. However, the presented formalization allows to get an estimate on the maximum population size for complex problems that are comparable to the real-world problem at hand. As the maximum population size only restricts the population size during the early learning phase, the estimate just has to be small enough to create enough selection pressure for proper evolution and large enough to allow exploration.[3] Section 5.4 presents a validation of the presented formalization.

4.5 Simulation library for functional and autonomic layers

This thesis contributed to the libslim and libasoc libraries of the ASoC project, with which the experiments of this thesis are carried out. The contributions include the separation of the external XML data model and the internal libslim model, several libasoc modules (modules for static and dynamic power estimation, time delay at a transistor, temperature-depending timing errors, and soft errors), an interface to HotSpot for temperature estimation, and an interface to the ANSI-C implementation of XCS (Butz, 1999). The internal libslim model allows programmatic manipulation of the architecture, for example to add logging modules or to enable or disable modules depending on flags given at the command line. The libasoc modules concerning physical parameters realize the models described in Section 2.3. The libasoc modules for static, dynamic and total power consumption implement the Equations 2.22 and 2.23, respectively. The libasoc module for temperature-depending timing errors implement the Golda model described in Equation 2.28, which is based on the models for the rise and fall time of a signal at an transistor described in Equations 2.26 and 2.25, respectively. The libasoc module for soft errors implements the Zhu model described by Equation 2.29. The libasoc module for temperature estimation implements an interface to the HotSpot tool (see Section 2.3.2). Besides the libasoc module that integrates the XCS, this thesis extends the XCS implementation of Butz (1999) to allow classifier migration between distributed XCS instances as described in Section 4.2.

An additional libasoc module allows the calculations of the HotSpot temperature model on a remote host. The *remote temperature module* allows to simulate SoC processing cores on physically different hosts while maintaining a global temperature distribution. The rationale is the following. Detailed simulation of a processing core may require large computational power. For detailed MPSoC simulations it may be prohibitive to run the simulation on a single physical core. Thus, it would be preferable that every simulated core could run on a different physical core, possibly on different hosts across a network. Unfortunately, the underlying simulation library SystemC does not support simulations

[3] Note that k_{avg} is in most cases much smaller than $k_{\max}$.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ASoCsim name="xcs-1" version="primitive-1.0">
  <Module name="sys" type="XCS_System">
    <Module name="CORE1" type="FE_traced">
      <param name="trace">
        <param name="file" value="core1.tr" type="path"/>
      </param>
      <port name="CACHE" target=":xcs-1.sys.L1CACHE_C1.IO"/>
      <Module name="PowerEst" type="C_totalPower">
        <param name="N_trans" value="59126896" type="int"/>
        <!-- further parameters  -->
        <depend target=":xcs-1.sys.CORE1.voltage" name="voltage"/>
        <depend target=":xcs-1.sys.CORE1.frequency" name="frequency"/>
      </Module>
      <!-- further modules of CORE1 -->
    </Module>
    <!-- further top level modules -->
  </Module>
</ASoCsim>
```

Figure 4.6: Excerpt of an architecture configuration file written for libslim and with modules defined in libasoc.

that run in multiple (host) processes, as synchronizing the communication of the different processes is very difficult. Parallelizing SystemC simulations was beyond the scope of this work, so the following approach was taken. The only property that has to be monitored globally in the following experiments is the temperature distribution of the MPSoC, as the temperature distribution depends on the energy consumption of every simulated core. Fortunately, the simulated cores need the current temperature value only in regular time intervals, namely after applying a parameter change and before calculating the reward. The time intervals are the same across all simulated cores. Thus, the simulated cores can repeatedly run asynchronously at first and then be synchronized when they need their current temperature. The synchronization is realized by the remote temperature module, which waits for each core to report its consumed energy before calculating the global temperature distribution and replying every core its new current temperature. Of course, this approach of alternating asynchronous and synchronous operation is only feasible if the simulation of direct or indirect inter-core communication is not necessary; otherwise, the simulation has to obey the single-thread property of SystemC simulations. To simulate the effects of changing environments, the remote temperature module allows to change the ambient temperature (and thus indirectly the temperature of the simulated cores) with a zigzag pattern of adjustable number of cycles, slope, lower bound, and upper bound.

Figure 4.6 shows an excerpt of a libslim architecture configuration file that uses modules defined in libasoc. It describes a module named xcs of type XCS_System that contains several further modules that represent the Opteron described in Section 5.1. One of these modules is a module named CORE1 of type FE_traced, representing one of the cores of the Opteron. Modules of type FE_traced are functional elements that execute a trace, a list of commands, which is stored in the file whose name is given by the parameter trace. The CORE1 Module has a port named CACHE which is connected to the input–output port of its L1-cache, addressed by :xcs-1.sys.L1CACHE_C1.IO. Modules such as PowerEst use several parameters to give accurate power estimations for the parent module; for example, N_trans describes the number of transistors of the parent module. The module description of PowerEst also states that it depends on the voltage and frequency monitor modules, which are not shown in the excerpt. By stating the dependency, the PowerEst module can be notified via an event whenever the voltage and frequency monitor modules change their values, so that PowerEst can update its power estimation. Similarly to CORE1, the configuration file contains descriptions of the other functional and autonomic elements of the design, which are not shown in the excerpt. This includes the other cores of the Opteron, its L2- and L3-caches, the northbridge, and the autonomic elements with their evaluators. Describing the architecture in an external configuration file is well suited for ASoC. For example, the designer can use a system with an XCS evaluator and, once satisfied with the results, just exchange the XCS evaluator for the actual run-time evaluator, using the classifiers that have been previously trained with the XCS. This approach is taken in the experiments of this thesis to show that the proposed methodology is realizable in hardware.

5 Experimental setups

This chapter describes the experiments with which the contributions of this thesis are analyzed and validated. The first three sections describe different application examples, while the remaining sections deal with further aspects of the proposed design methodology. The experiments presented in Section 5.1 apply the proposed design methodology to an AMD Opteron, a quad-core multi-purpose CPU. The section describes the simulation parameters of the AMD Opteron, three different scenarios in which the self-adaptation is tested as well as the related reward functions. The experiments are supposed to show that the proposed self-adaptive is applicable to control the operating point of a MPSoC and that it self-adapts to unforeseen events.

The experiments in Section 5.2 apply the proposed design methodology to a simulation model, whose physical properties are oriented at the Cell processor. The section describes the MPSoC simulation model, the various parameter settings, the reward function that is used, and how the experiments are evaluated. The experiments are supposed to show the scalability of the self-adaptive controller and that self-adaptation with cooperation solves problems that self-adaptation without cooperation cannot handle. The experiments extend the current state of the art of distributed XCS: unlike in previous work, here every XCS instance solves a slightly different problem, as the cooling properties of the multiprocessor cores differ from each other.

The experiments described in Section 5.3 apply the proposed design methodology to the operator-allocation problem. The XCS is set to solve the operator-allocation problem, to self-adapt to an unforeseen event and to generalize after it has learned only on a subset of the possible input space. The experiments are supposed to show that the self-adaptive controller is also applicable to combinatorial problems (and not only to control problems), its benefits compared to static lookup tables, and its capability to generalize from restricted training sets.

Section 5.4 explains the experimental setup to validate the extended XCS theory by using the operator-allocation problem. The experiments are supposed to show that the theoretically derived maximum population sizes are upper bounds, that is, larger population sizes are not necessary while smaller population sizes diminish the performance, which indicates classifiers that are less than optimal. As the maximum population size is proportional to the problem dimension, the experiments will also show indirectly whether the problem dimensions are estimated correctly.

Section 5.5 describes the experimental setup to analyze the effects of the proposed method of subsuming after learning, namely the resulting size of the classifier population, the performance of the subsumed classifiers in the operator-allocation problem, and the ability of the subsumed classifiers to self-adapt to an unforeseen event.

The experiments described in Section 5.6 show that the proposed design methodology

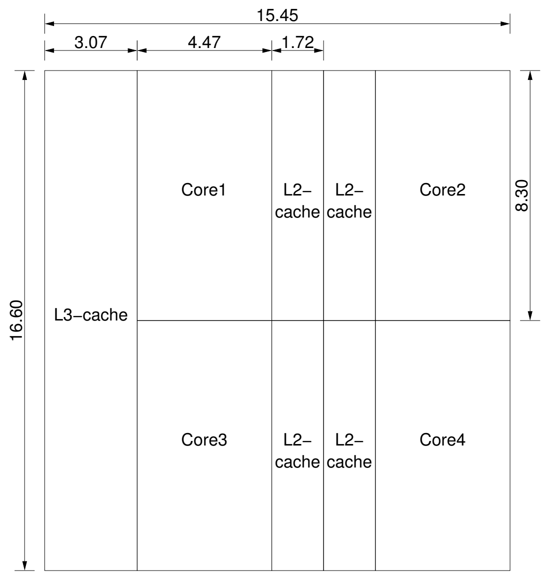


Figure 5.1: Floor plan of the Opteron. Measurements are given in mm.

can realize a self-adaptive controller in hardware. The hardware implementation is supposed to solve the academic multiplexer problem and the operator-allocation problem at run time.

5.1 Controlling a multi-core chip

A first set of experiments investigates whether the proposed self-adaptive controller is actually able to control the operating point of a SoC and whether it exhibits its self-adaptive property.

This section describes the hardware and software used for the evaluation of the XCS, the encoding of the environment and actions of the XCS, and the tests of the evaluation.

As a multi-processor SoC (MPSoC) hardware, the AMD Opteron (Barcelona) processor is chosen, which is a four-core general purpose processor produced on a single die. The advantage of the Opteron as an MPSoC is that most of the parameters which are necessary for simulation are publicly available and that the processor can adjust the frequency of each core individually. Figure 5.1 shows the floor plan of the Opteron, as derived from Opteron's die shot[1]. The activity factor of the cores is adjusted so that the thermal design power and average CPU power are met. Each core is controlled by one XCS which is assumed to run on dedicated hardware, so regular core operation is not interrupted.

Four algorithms are executed on the hardware: LR-decomposition, video filtering, matrix multiplication and a dual-process application where one process has to wait for the other. The algorithms are executed with traces, which only describe the memory

[1] The Opteron die shot appears in `http://developer.amd.com/assets/CART2007-Barcelona.pdf` (retrieved October 18, 2010).

access patterns, needed computation time and the activity factor without computing anything actually. This decreases simulation time while still allows good estimates. When all cores execute an algorithm, power consumption lies at 83 W and temperature at 55 °C. When all cores are idling, power consumption lies at 26 W. These simulated values are comparable to the actual values of the Opteron[2].

Eleven different frequencies (500–3000 MHz, encoded in four bits) and five voltage levels (0.8–1.3 V, encoded in three bits) are used. Temperature range is 50–90 °C, encoded in five bits. The action sets the frequency and the voltage. For this evaluation, frequency and voltage are treated separately; a combined usage would have been possible, too.

Given a trained XCS, three scenarios are evaluated: simple control, control under changed environment, and online learning without genetic algorithm. In the simple-control scenario, the XCS should find the optimal operating point under training conditions (described later). For this, every 10 s of simulation time a random frequency and voltage is set and the XCS is supposed to reset the frequency and voltage to the optimal operating point. In the changed-environment scenario, the XCS should find the optimal operating point although the ambient temperature is raised by 15 K and the reward function is changed from its setting during training. The online learning scenario evaluates whether the XCS can learn a new reward function without its genetic algorithm, as on a SoC the genetic algorithm isn't expected to be available.

All scenarios are modeled as single-step problems of the XCS, as for a multi-step problem the optimal operating point generally has to be known in advance (to signal the end of the problem to the XCS and distribute the reward) (Butz and Wilson, 2001). To simulate the situation of the XCS on a SoC, where there is no genetic algorithm and no new classifiers are created, exploration is inhibited ($P_{\text{explr}} = 0$) and the genetic algorithm is turned off ($\theta_{\text{GA}} = \infty$).

The XCS is trained on a single core with an activity factor of 0.05 without simulating cache access. The other cores are idling at 2000 MHz at 1.2 V. This setup allows a significantly smaller simulation time than running the algorithms. A single run consists of 50 000 repetitions, until all possible frequency-voltage pairs are tested sufficiently often. Between the runs, temperature is raised randomly by 5 K, 10 K, 20 K, or 30 K from default. For the online-learning scenario, $\beta = 1.0$ to reduce the time needed for learning, otherwise, the default values are used.

Reward functions

The reward function reflects that the XCS should maximize performance and minimize power consumption, while keeping the error rate low. This results in the following reward function for the training phase and the simple-control scenario:

$$R(f, p, t, v) = w_1 \frac{f}{f_{\max}} + w_2 \left(1 - \frac{p}{p_{\max}}\right) + w_3 \mathrm{rel}(t, v, f) \tag{5.1}$$

[2] See AMD Whitepaper on Power Consumption, Publication 43761A, published at `http://www.amd.com/us-en/assets/content_type/DownloadableAssets/43761A_ACP_WPv7.pdf` (retrieved January 13, 2008). Opteron product sheets are also published at `http://www.amd.com/us/products/server/processors/opteron/Pages/opteron-for-server.aspx` (retrieved October 18, 2010).

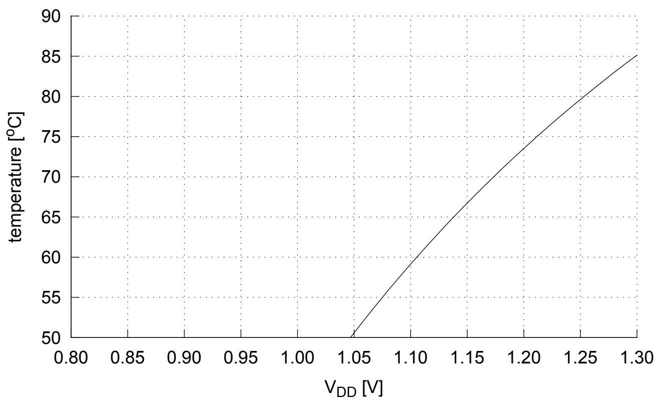


Figure 5.2: Maximum temperature for timing-error–free operation at 2000 MHz, depending on voltage.

Here, $\mathrm{rel}(t, v, f)$ models the reliability and is 0 in case of an error and 1 otherwise. $w_1 = 200$, $w_2 = 35$, and $w_3 = 200$ are selected as the relative weights of the optimization goals. Figure 5.2 shows the fault count depending on temperature and voltage at 2000 MHz. It can be observed that [2000 MHz, 1.2 V] is a safe setting in terms of faults at a usual temperature range (up to 70 °C).

In the scenario with changed environment, the $\mathrm{rel}(t, v, f)$ function in Equation 5.1 is changed to

$$\mathrm{rel}(t, v, f) = \begin{cases} 0 & \text{if timing error} \\ (\frac{70}{t})^2 & \text{if } t > 70 \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

and the weights are changed to $w_1 = 200$, $w_2 = 35$, and $w_3 = 200$ if the temperature is below 70 °C and $w_1 = 100$, $w_2 = 100$, and $w_3 = 200$ if the temperature is above 70 °C. This represents an emergency behavior which allows the XCS to use a less performing setting and shows how the designer's prior knowledge may enter the XCS control mechanism.

In the scenario with the genetic algorithm disabled, the goal of the reward function is changed to also minimize the waiting time between two processes:

$$R(f, p, t, v, w) = w_1 \mathrm{time}(w) + w_2 \left(1 - \frac{p}{p_{\max}}\right) + w_3 \mathrm{rel}(t, v, f) \tag{5.3}$$

Here,

$$\mathrm{time}(w) = \begin{cases} 1 - \frac{w}{w_{\max}} & \text{if the waiting time of Core 1 is zero} \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

and $\mathrm{rel}(t, v, f)$ is the same as in Equation 5.2.

5.2 Controlling a multi-core chip using cooperation

This section describes the simulation model that is used to evaluate and analyze distributed XCS, the parameter settings of the simulation model, and the experimental procedure.

While in the experiments of the previous section there is little interaction between the cores, high interaction between the cores may render the control problem unfeasible for the XCS. The experiments of this section are supposed to show the scalability of the proposed self-adaptive controller. No higher-level controller is necessary; instead one cooperating, self-adaptive controller per additional component is sufficient. The following experiments simulate an MPSoC that has a high thermal interaction between the cores; that is, the heating of a core affects its neighboring cores. For example, this situation can be observed in the Cell Processor (Pham et al., 2005c). If the temperature of a core rises above an (unknown) threshold, the timing in the critical path is violated and a timing error occurs. The XCS instance of each core is supposed to adjust the core's frequency and voltage such that no timing errors occur, but the performance (i.e., the frequency) is as high as possible. An in-advance experiment will show that isolated, non-cooperating XCS instances are not able to avoid timing errors.

5.2.1 Simulation model

The experiments use the Cell processor of the first generation (Pham et al., 2005a) as a basis for the simulation model. The Cell processor is very versatile: it is used as PowerXCell 8i in high performance computers of IBM, as a processor in the game console Playstation 3 of Sony and as an image processor in television sets of Toshiba. Figure 5.3 shows a schematic diagram of the Cell processor. The Cell processor consists of 234 million transistors implemented in 90 nm silicon-on-insulator technology (Pham et al., 2005b) on 221 mm^2 and comprises several cores, which can be configured independently. The SoC contains a Power Processing Element (PPE), which is a regular PowerPC core consisting of the actual core (PXU), and the L1- and L2-caches. The PXU knows simultaneous multi-threading and out-of-order execution. Along the PPE, there are eight Synergistic Processing Elements (SPE) consisting of the actual core (SXU) with in-order execution and a local memory (LS) of 256 kB. The SoC cores integrate their own memory and communicate with a ring bus (Element Interconnect Bus, EIB). The chip contains an external memory interface (Memory Interface Controller, MIC) and an input–output interface (FlexIO). It contains 15 different voltage zones and ten temperature sensors. Under laboratory conditions, the chip can be operated between 900 mV and 1500 mV and 3000 MHz and 4400 MHz if its temperature is fixed at 85° C.

Simplified simulation model

Although the computational load of simulating the cores of the MPSoC can be spread across several physical cores, the calculation of the global temperature distribution still has to happen on a single core (because the HotSpot temperature simulation is single-threaded) and hence is the bottleneck of the simulation. As the time that the remote

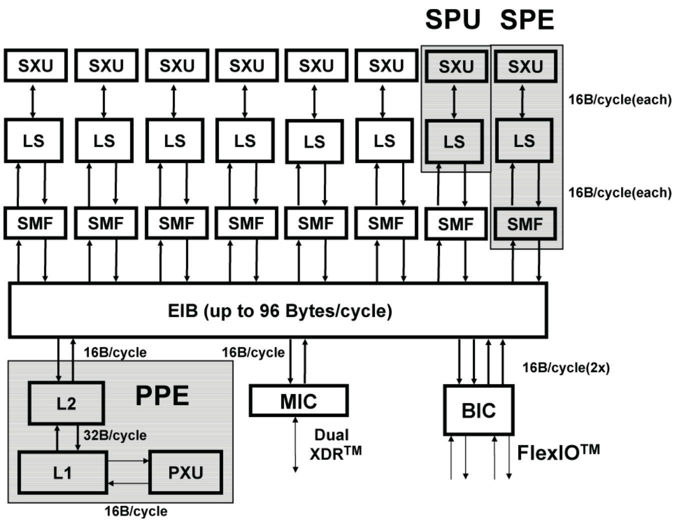


Figure 5.3: Schematic diagram of the Cell processor. From Pham et al. (2006).

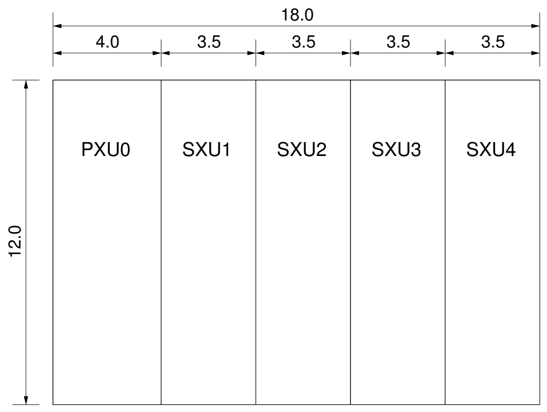


Figure 5.4: Floor plan of the simulated MPSoC. Measurements are given in mm.

temperature module needs to calculate the global temperature distribution directly depends on the number of simulated cores, the MPSoC simulation model is simplified such that one simulation run finishes within 24 hours. Only the cores (PXU and SXUs) are simulated, ignoring bus and interfaces (see the floor plan depicted in Figure 5.4). Opposite SXUs are pooled so that there are only four SXU blocks, minimizing the total time for temperature simulation. The transistors of the caches and memories are added to the simulated cores, keeping total transistor count constant. It is assumed that the transistor density is 10^6 per square millimeter as derived from the Cell data sheet, so that the transistor count of each block can be derived by its area. With this, each (pooled) SXU block consists of 42 million transistors and the PXU block consists of 48 million transistors, including its caches. In total, this makes 216 million transistors, which is close enough to the 234 million transistors given in the literature (Pham et al., 2005b), considering that the FlexIO and the MIC are ignored (as they do not considerably contribute to the global temperature distribution as can be seen in the thermal map in Pham et al. (2005c)).

Reward function

The reward that the XCS receives depends on the effects of the actions that the XCS has selected. After the environment receives the action from the XCS, it applies the chosen frequency and voltage settings. The environment checks twice for timing errors: immediately after setting the new frequency and voltage with the current temperature and after the synchronization time interval has passed with the updated temperature. In the experiments, the synchronization time interval is three seconds, so that temperature can spread across the MPSoC. As during the synchronization time interval the temperature changes only monotonically most of the time, the two checks ensure that the environment catches virtually all timing errors. The reward depends on whether the environment detected the timing error immediately (*immediate timing error*) or only during the synchronization time interval (*delayed timing error*), allowing the reward function to better hint towards the desired optimum (*layered payoff*).

Layered payoff in the form of a stepwise reward function is used, whose parts point into the direction of the desired optimum (high performance, no timing error). It ensures that any action that results in a timing error is rewarded less than an action that resulted in no timing error. Additionally, to promote higher performance, it rewards higher frequencies (for simplicity; a more sophisticated performance measure could be used instead). Thus, the reward function is both encouraging higher performance and pointing into the direction of timing-error–free operation. In the case of an immediate timing error, the reward function returns the selected frequency as a reward. In the case of a delayed timing error, the reward function returns twice the selected frequency as a reward. In the case of no timing errors, the reward function returns four times the selected frequency as a reward, clearly distinguishing the error-free case from the erroneous cases. Hence, the reward function is calculated as follows (compare Figure 5.5):

$$r(f) = \begin{cases} 1 \cdot f & \text{if immediate timing error} \\ 2 \cdot f & \text{if delayed timing error} \\ 4 \cdot f & \text{if no timing error} \end{cases} \tag{5.5}$$

The reward function ensures that the three resulting plateaus do not overlap, so that the direction towards the desired goal is clear for the XCS. While the design of the reward function is supposed to help the XCS to find the optimal setting, the reward function is disadvantageous if no error-free setting is possible (e.g., if the ambient temperature is too high), as even in the erroneous case it promotes higher frequencies. At first sight, negating the slope of the lower plateau (the case of immediate timing errors) should help alleviate this problem. However, a preliminary analysis showed detrimental effects. Therefore, the following assumes that at least one setting allows error-free operation under any circumstances (here: low frequency, high voltage).

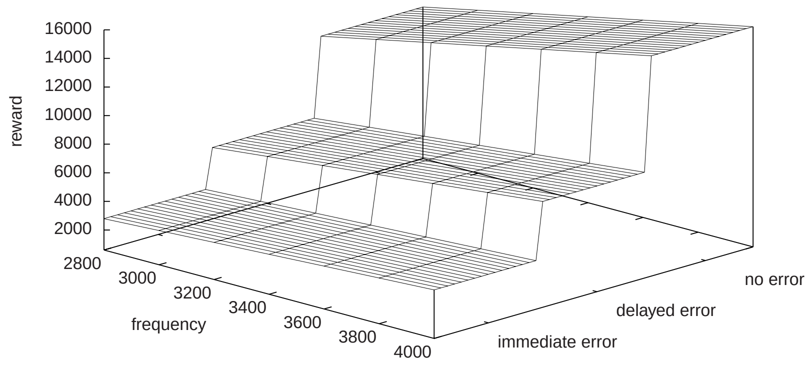


Figure 5.5: Reward function with three plateaus in the simulation of the MPSoC.

Physical parameter settings

The frequency of each simulated core can be adjusted in eight steps of $f_{\text{step}} = 200\,\text{MHz}$ between $f_{\text{min}} = 2600\,\text{MHz}$ and $f_{\text{max}} = 4000\,\text{MHz}$. For the voltage, there are two levels, 900 mV and 1150 mV. The temperature is quantized linearly in 5 bits. The limits for the core temperatures are 50° C and 95° C and for the ambient temperature 20° C and 60° C. These temperatures constitute the monitor input to the XCS of length $L = 10$.

The XCS selects one of 16 possible actions $a \in A = \{0, 1, \ldots, 15\}$, where the first three bits represent the step factor f_s for the frequency and the last bit describes the voltage level. The final core frequency is $f = f_{\text{min}} + f_{\text{step}} \cdot f_s$.

XCS parameter settings

Table 5.1 summarizes the parameter settings for the XCS. Section 2.1 explains the parameters number of actions n, maximum population size N, maximum number of steps N_{steps}, don't-care probability $P_{\#}$, and threshold ε_0 of the relative prediction error. The number of actions $n = 16$ results from the eight frequency and two voltage levels. The maximum number of steps $N_{\text{steps}} = 14\,000$ is an empirically derived value and suffices to observe the relevant properties of the experiments. The maximum population size $N = 6\,000$ is sufficiently large so that the population can keep at least five different actions for any of the $2^{10} = 1024$ possible temperature readings (this is an empirical rule of thumb). The values for the don't-care probability $P_{\#}$ are mostly taken from the literature (Butz et al. (2001) report 0.1 to be most beneficial, 0.3 is the default value in the C-implementation of XCS (Butz, 1999), 0.6 seems a sensible intermediate value, and 0.9 will be shown to be beneficial in the first application example).

The parameters $\delta_{\text{emigrants}} = 0.005$ and $\delta_{\text{hold}} = 0.001 N_{\text{steps}}$, which means that at most 30 classifiers emigrate per explore step and migration starts after the 14. step. The migration strategy $\mathcal{M}$, the topology $\mathcal{T}$ and the deletion strategy $\mathcal{D}$ take the values as previously explained, except that deletion by action set size (deletion strategy A) is excluded as preliminary analysis showed that the results of deletion by action set size are of limited value and this finding is taken as an opportunity to reduce the overall time needed to run the simulations.

Table 5.1: Parameter settings of the XCS in the simulation of the MPSoC.

Parameter	Value
n	16
N	6 000
N_{steps}	14 000
$P_{\#}$	$\{0.1, 0.3, 0.6, 0.9\}$
ε_0	0.001

Parameter	Value
$\delta_{\text{emigrants}}$	0.005
δ_{hold}	$0.001 N_{\text{steps}}$
$\mathcal{M}$	$\{R, N, F\}$
$\mathcal{T}$	$\{U, B, C\}$
$\mathcal{D}$	$\{N, F\}$

Table 5.2: Values of the physical parameters in the simulation of the MPSoC.

Parameter	Value
k_{design}	11
I_{leak}	10 nA
C_{trans}	1.5 fF
chip width	0.15 mm
T_{start}	60° C
C_{convec}	$140.4\,\frac{\text{W}}{\text{K}}$
R_{convec}	$0.1\,\frac{\text{K}}{\text{W}}$

Parameter	Value
l_{sink}	5 cm
w_{sink}	6.9 mm
l_{spreader}	3 cm
w_{spreader}	1 mm
β	$115\,\frac{\mu\text{S}}{\text{V}}$
$V_{\text{tn}} = V_{\text{tp}}$	200 mV
$t_{\frac{d}{\text{K}}}$	0.5 ps
N_{inv}	6

The physical parameters besides the transistor count that are used for the models of the simulation are given in Table 5.2.

5.2.2 Experimental procedure

The experiments run as follows. For every possible combination of migration strategy $\mathcal{M}$, deletion strategy $\mathcal{D}$, topology $\mathcal{T}$, and don't-care probability $P_{\#}$ there are three runs with different random seeds. Every run consists of two parts, a training part and an application part, each of which runs for 42 000 simulated seconds (i.e., 14 000 steps à three seconds synchronization time interval). The cores observe an artificial activity distribution with periods of constant $\alpha \in \{0.05, 0.1, 0.15, 0.20\}$ that last between 0.5 s and 3.3 s, depending on the current frequency settings. The average activity is $\alpha = 0.1$. During the training part, the XCS explores every other step; that is, $P_{\text{explr}} = 1$. During the application part the XCS takes no explore steps and acts on knowledge that it has acquired in the training part; that is, $P_{\text{explr}} = 0$. Note that learning still happens in the application part, mainly by adjusting the parameters of the classifiers such as fitness, prediction error, and accuracy. The separation into an training part and an application part allows to accurately assess the learning of the XCS. Otherwise, the intermediate explore steps, where the XCS can take random actions, disturb the chip setting too much for accurate measuring. Besides, the application part is the intended mode of operation of the XCS.

All presented results are averages over the application parts of the three runs.

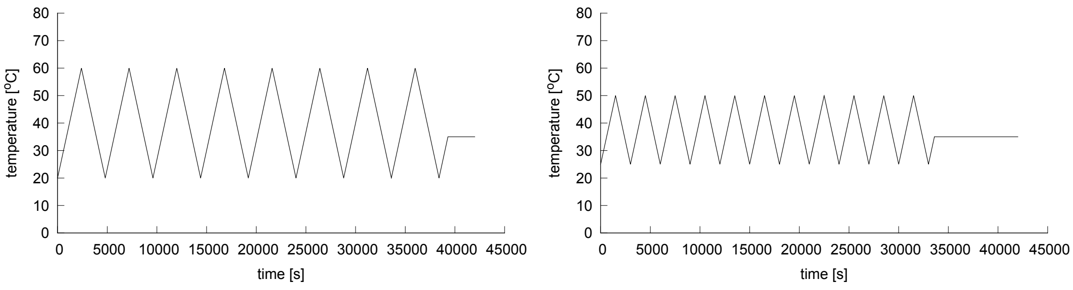


Figure 5.6: Course of ambient temperature during the training (left) and application (right) part of the simulation of the MPSoC.

Ambient temperature

The environment changes the ambient temperatures repeatedly (see Figure 5.6). In the training part, the ambient temperature changes in eight cycles between 20° C and 60° C, changing by 0.05 K at each step (of three seconds). In the application part, the ambient temperature changes in eleven cycles between 25° C and 50° C with the same slope. The lower bound of 20° C approximately corresponds to the ambient temperature upon power-on. The upper bound of 60° C in the training part clearly exceeds the usual ambient temperature of 45° C used in simulations as reported by Skadron et al. (2003). However, preliminary analysis shows that at the maximum ambient temperature, an error-free operation is possible, fulfilling the assumption of the reward function (Section 5.2.1). Repeating the temperature cycles and avoiding the edge cases in the application part ensures a proper assessment of the XCS instances within the usual temperature interval.

Evaluation of experiments

The training parts of the experiments are evaluated according to the following measures:

1. The *average prediction error* shows how well the XCS instances can predict the outcome of their actions.

2. The *average success rate* or the average error rate per temperature cycle and core show how well the XCS instance can avoid timing errors. The average success rate is one minus the average error rate.

3. The *performance* is measured as the frequency that the XCS achieves.

On the long run, the average prediction error should reach zero and the success rate should reach one. In the ideal case, the XCS instance starts the application part with error-free action selection, as the necessary classifiers should have been learned in the training part. However, due to the disturbing effects of the explore step in the training part, a grace period is expected, in which the XCS sorts out detrimental classifiers.

5.3 Solving the combinatorial operator-allocation problem

This section describes the experimental setup that is used to apply the proposed design methodology for a self-adaptive circuit to combinatorial problems. The section uses the combinatorial problem of operator-allocation as presented in Section 4.3.

The unforeseen event for the operator-allocation problem is an unmonitored operator failure, that is, although an operator has failed, it is monitored as free and the XCS cannot allocate it. The generalization capabilities are evaluated by training the XCS only on a fixed subset of the possible monitor input space but evaluating it over the whole input space.

The first set of experiments represents the design-time learning of the evaluator and shows the basic performance level that the 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 XCS learns based on only 25% and 5% of all possible operator occupations to reduce the time needed to complete the design of the self-adaptation system. All experiments are performed as software simulations.

The experiments run as follows. During each learning step, a random number r of the c operators 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 $(10, 3)$ allocation problem, there exist $2^{10} = 1024$ different operator 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 (*action zero*) 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.

For each simulation run of the XCS, the results show the average values of the last 500 learning exploit steps, as usual (Butz, 1999). All experiments are run with the default values except for the following changes: $N = 10\,000$, $\gamma = 0.8$, $\theta_{\mathrm{GA}} = 250$, $\xi_{\mathrm{GA}} = 0.1$, $\mu_{\mathrm{GA}} = 0.1$, $P_{\#} = 0.4$, $P_{\mathrm{explr}} = 0.2$, with activated generalization mutation, Moyenne Adaptive Modifiée (MAM), GA subsumption, and action-set subsumption. The parameters are chosen to balance keeping acquired knowledge and discovering new classifiers.

The *correctness rate* is quantified and a relative measure of the *chip area* of the evaluator is estimated. 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%. The correctness rate will be used to determine the performance of the evaluator in the basic version and how well it can self-adapt. The number of bits that are necessary to store the evaluator's knowledge is used 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 the self-adaptation system, other systems are needed for comparison. However, as the purpose is to reduce design effort, systems that have been specially designed for the operator-allocation problem cannot be considered, as this would counteract the intentions of this work. Therefore, the following two evaluators are considered for comparison: *random allocation* and *lookup table*. Both evaluators fit the interface presented in Figure 4.1 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 correctness rate of the basic version. 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 correctness 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 self-adaptation system, they fulfill the requirement that they are not specially designed for the evaluation problem.

5.4 Validating the extended XCS theory

As mentioned in Section 4.4, in the operator-allocation problem the number and structure of the optimal classifiers vary considerably and, in particular, the problems show a large variation in the number of relevant bits across the problem instances. The different operator-allocation problems thus represent a large variety of a class of complex problems, which is why they are chosen to validate the extended XCS theory.

To validate the extended XCS theory, the formulation of an equivalent problem dimension is applied to all (c, t) operator-allocation problems with $1 \leq t \leq c \leq 10$. For each operator-allocation problem, the following steps are performed. First, the minimum set of classifiers that correctly predict the reward for every possible state and action is considered. Following the development of the XCS theory on regular problems, this minimum set is called *optimal classifiers*. Then, Equation 4.3 is applied to get the equivalent problem dimension k_{avg}. After k_{avg} is known, it is used to deduce the maximum population size N and the corresponding don't-care probability $P_{\#}$ for each operator-allocation problem such that the reproductive opportunity is taken (Equation 2.19) and the covering and schema challenges (Equations 2.14 and 2.16, respectively) are met. Last, the XCS is set to solve each operator-allocation problem with the corresponding maximum population size N as well as $0.75N$ and $0.5N$. The resulting correctness rates (i.e., the ratio of valid allocations suggested by the XCS) are recorded for each problem. The XCS parameters are set as usual for the operator-allocation problem: $\theta_{\text{GA}} = 250$, $\mu_{\text{GA}} = 0.1$, $\chi_{\text{GA}} = 0.5$, action set subsumption and GA subsumption are turned off, N and $P_{\#}$ have their previously calculated, problem-specific values, and all other parameters are left at their default value.

A correctness rate of over 90% is considered as good enough for practical purposes, as the self-adaptation system is not supposed to always propose a perfect solution. The challenges are thus assumed to be met if the probabilities $\Pr(\text{cover})$ and $\Pr(\text{representative})$ are greater than 90%, respectively. Although the maximum population size N and the corresponding don't-care probability $P_{\#}$ could be calculated directly, an exact solution for $P_{\#}$ is not necessary, as the XCS is known to not be very sensitive to the choice of $P_{\#}$ (Butz et al., 2001; Studley and Bull, 2005). Thus, the following simpler approach is taken. The smallest value for the maximum population size is searched that takes the reproductive opportunity and meets the two challenges using the largest possible don't-care probability from the set $\{0.2, 0.3, \ldots, 1.0\}$. Values of $P_{\#} < 0.2$ are known to result in poor learning performance (the initial classifiers are over-specific) and are thus avoided.

5.5 Subsuming after learning

The operator-allocation problem is also used to analyze the effects of the proposed method of subsumption after learning. The following is analyzed: the resulting size of the classifier population, the performance of the subsumed classifiers in the operator-allocation problem, and the ability of the subsumed classifiers to self-adapt to an unforeseen event. As the classifiers that are subsumed, the classifiers from the previous operator-allocation experiments (Section 5.4) are used.

The population sizes of the subsumed classifiers are compared to population sizes of experiments where subsumption is not used, of experiments where subsumption is used during learning and the number of optimal classifiers. Then, the XCS is rerun with the newly created population of subsumed classifiers and the resulting correctness rates are recorded as averages of the last 50 exploitation steps. The discovery component is turned off in these experiments; first, because the hardware implementation is not expected to include a discovery component and, second, as the classifiers are assumed to be optimal, they do not need to be challenged by newly introduced classifiers. A final experiment is used to analyze whether the XCS is still able to self-adapt if it uses the subsumed classifiers. For this, the XCS with the subsumed classifiers is exposed to the unforeseen event of operator failure, as described in Section 5.3, and the resulting correctness rates are recorded.

5.6 Running in hardware

In contrast to the previous experiments, the final set of experiments executes a possible hardware realization of the proposed self-adaptive controller using the LCT. As there is no method to generate an appropriate rule table for the LCT, two more ways to generate LCT rules are considered, full-constant and full-reverse, to compare the proposed methodology with the base performance of the LCT. Both translations provide all possible LCT rules, that is, a complete condition-action table[3]. The full-constant translation initializes the rule fitness to half the maximum reward (in the experiments: 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

[3] Of course, the memory requirements of the classifiers generated with full-* grow exponentially with the problem size. They are only used for comparison.

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 methodology is assessed with two problem types, the multiplexer problem (Wilson, 1994) and the operator-allocation problem. Additionally, an unforeseen event is defined for each problem type to explore the learning ability of the hardware implementation: the LCT has to handle an unforeseen event for which the XCS had no chance to learn classifiers. As the XCS has already been shown to be able to solve these problem types and adapt to unforeseen chip events, these experiments concentrate on the performance of the LCT.

The unforeseen event for the multiplexer problem is the *inversed multiplexer*: instead of the value of the indexed bit, the LCS is supposed to return the *inversed* value of the indexed bit. For example, in the inversed 6-multiplexer problem, $\overline{m}_6(011101) = 1 - m_6(011101) = 1$. The same XCS parameters are used as in the full-fledged FPGA implementation of XCS presented by Bolchini et al. (2006) to have comparable results: $\beta = 0.2$, $\delta = 0.1$, $\varepsilon_0 = 10$ (which is 1% of the maximum reward), $\nu = 5$, $\theta_{\text{GA}} = 25$, $\chi_{\text{GA}} = 0.8$, $\mu_{\text{GA}} = 0.04$, $P_{\#} = 0.3$; GA subsumption is on with $\theta_{\text{GAsub}} = 20$, while action set subsumption is off. Generalization or niche mutation is also turned off. The reported results are averages over 20 runs.

The unforeseen event for the operator-allocation problem is the unmonitored failure of an operator, as already mentioned in Section 5.3. For the operator-allocation problem, the XCS parameters from Section 5.3 are used to have comparable results, which differ from the multiplexer settings only in the following parameters: $\theta_{\text{GA}} = 250$, $\chi_{\text{GA}} = 0.1$, $\mu_{\text{GA}} = 0.1$, $P_{\#} = 0.4$; GA subsumption is off. The reported results for the operator-allocation problem are averages over 5 runs, a smaller number due to the longer simulation time for the many problem instances.

The LCT is simulated with a libasoc-based simulation model of its hardware implementation[4], with the additional winner-takes-all strategy described in Section 4.1.2. While the XCS is usually configured to alternate between the explore and exploit mode, in the experiments the LCT uses only one of either strategies.

The performance of the LCT that has been instructed by the proposed methodology is compared with the base performance of the LCT, the performance of the full-fledged hardware implementation of the XCS presented by Bolchini et al. (2006), the performance of the XCS from the experiments of Section 5.1, and the performance of the software version of the XCS. Furthermore, it is checked whether the LCT retains the capability to adapt to unforeseen events.

[4] I thank Johannes Zeppenfeld for providing me his implementation of the LCT.

6 Results

This chapter presents the results of the experiments with the proposed design methodology and an estimate on the involved costs. The experimental setups are described in the previous chapter.

The chapter is structured analogously to the previous chapter plus an additional section on the assumed costs of the proposed design methodology. Section 6.1 presents the results of applying the proposed design methodology on an AMD Opteron. Section 6.2 presents the results of applying the proposed design methodology on a multi-processor SoC with cooperating self-adaptation units. Section 6.3 presents the results of solving the operator-allocation problem and the generalization capabilities of the self-adaptation unit. Section 6.4 presents the results on validating the extended XCS theory. Section 6.5 presents the results on subsuming after learning. Section 6.6 presents run-time results of applying the proposed design methodology to solve the multiplexer problem and the operator-allocation problem. Section 6.7 presents estimates on the costs of applying the proposed design methodology.

6.1 Controlling a multi-core chip

This section presents the results on applying the proposed design methodology to the model of the AMD Opteron as described in Section 5.1. The results are published in Bernauer et al. (2008).

Figure 6.1 shows the resulting frequency and voltage settings in the simple-control scenario. The figure shows that the XCS continuously resets the frequency and voltage to 2000 MHz and 1.2 V after the random disturbances, which leads to no errors in the actual temperature range (see Figure 5.2) and is the optimal setting. Each XCS contains about 600 rules, which would require about 8 kB (104 bit per rule including 84 bit solely for rule parameters such as fitness, etc.) if it is implemented unmodified on the SoC.

Figure 6.2 shows the result for the changed-environment scenario. It shows that once the temperature raises above 70 °C due to an external change in the ambient temperature, the XCS changes the frequency and voltage such that temperature falls again and timing errors stay low. However, once the temperature drops below 70 °C, the XCS resets the frequency and voltage again to a higher setting, causing the system to oscillate until the ambient temperature is reset to its usual value.

For the online learning scenario, Figure 6.3 shows the frequency-voltage pairs that the XCS tries out while learning the new reward function, which aims to minimize the waiting time w of Core 2 for Core 1 (and thus keep total run time low). The figure shows that, despite the high learning rate, the XCS needs a long time to learn the new reward

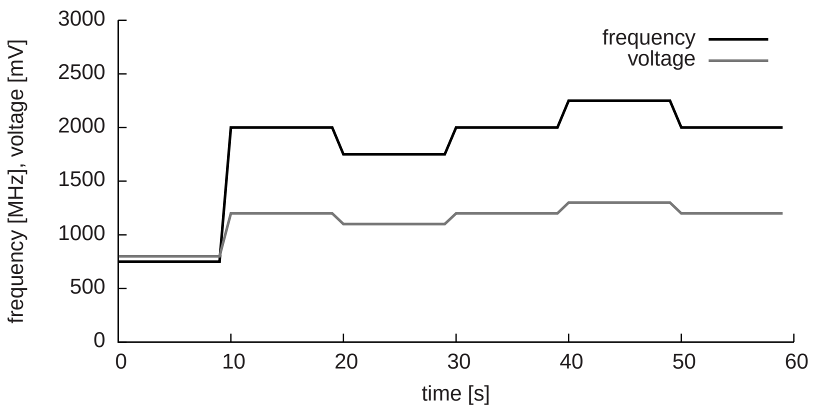


Figure 6.1: Control reaction of the XCS after setting a random frequency and voltage every 10 s.

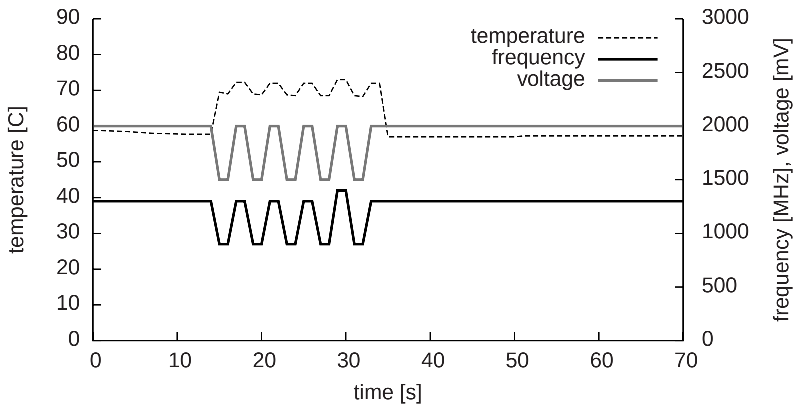


Figure 6.2: XCS' behavior to a rise in the ambient temperature of 15 K at $t = 15$ s for 16 s.

function. However, and most importantly, the figure also shows that the XCS is able to self-adapt to the new reward function.

6.2 Controlling a multi-core chip using cooperation

This section presents the experimental results for controlling the multi-processor SoC described in Section 5.2 with distributed, cooperating XCS instances. The data has been generated in the supervised diploma thesis of Arndt (2010). Section 6.2.1 presents a preliminary analysis of the simulation model without any XCS instances and shows that the assumption of the reward function (that an error-free operation is possible at any ambient temperature) is fulfilled. Section 6.2.2 presents the baseline results of isolated, non-cooperating XCS instances. Then, the following sections show the results for distributed XCS instances that select their emigrants at random (Section 6.2.3), by numerosity (Section 6.2.4), and by fitness (Section 6.2.5). Finally, Section 6.2.6 shows

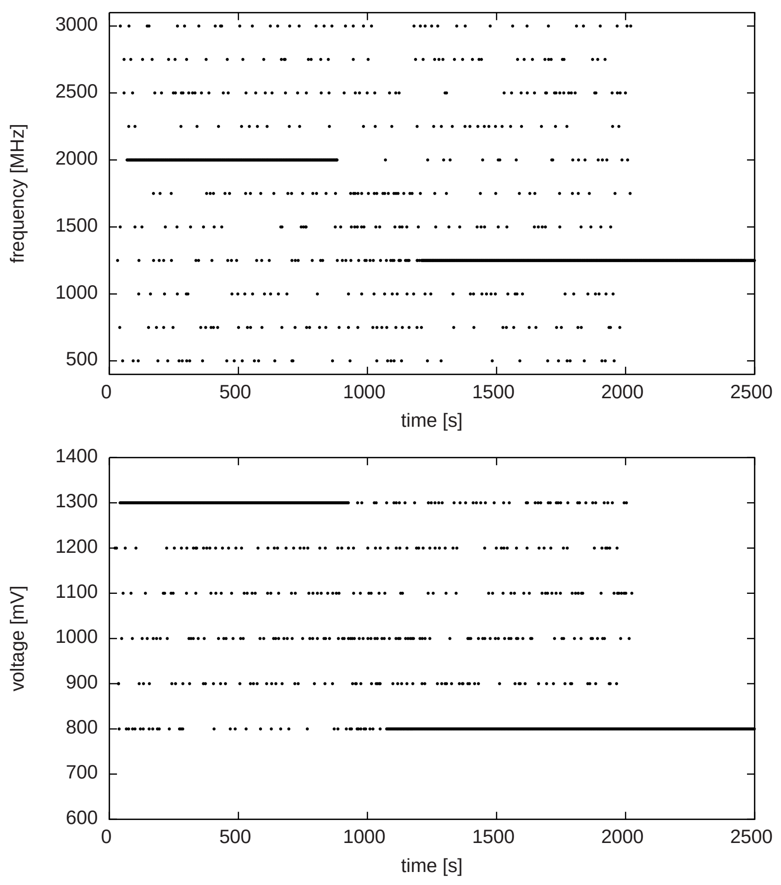


Figure 6.3: XCS learning a different reward function with disabled genetic algorithm and initial classifiers. Each point represents a frequency or voltage setting the XCS tries out. After about 2200 s this exploration mode is turned off, forcing the XCS to always choose the action with the highest prediction.

how the distributed XCS instances can adapt to different environments. Because a preliminary analysis of XCS instances that select their emigrants by action set size has not been convincing and because emigration by fitness shows satisfactory results, analysis of emigration by action set size is not pursuit further.

6.2.1 Preliminary analysis without XCS

First, an in-advance study ensures that, in the present simulation model, an error-free operation is always possible, so that the reward function is applicable. The ambient temperature is 20° C, 40° C, or 60° C. The frequency ranges from 2 600 MHz to 4 000 MHz. Only the results for the frequency settings where an error-free operation is still possible or not anymore possible are presented. The in-advance study is performed with maximum activity, resulting in maximum temperatures.

Figure 6.4 shows the temperature course of all cores for 20° C ambient temperature at

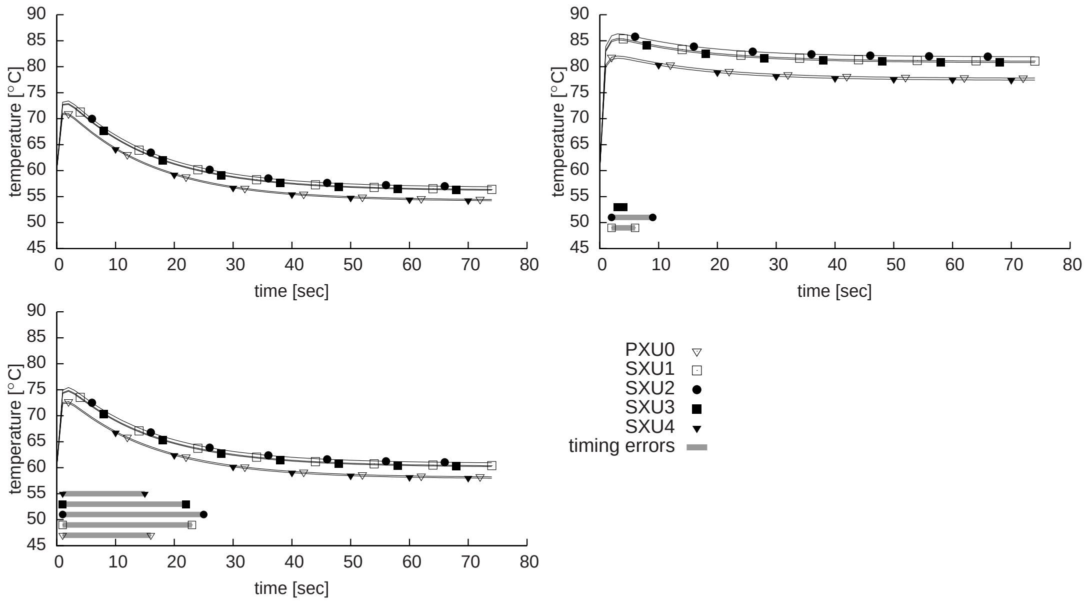
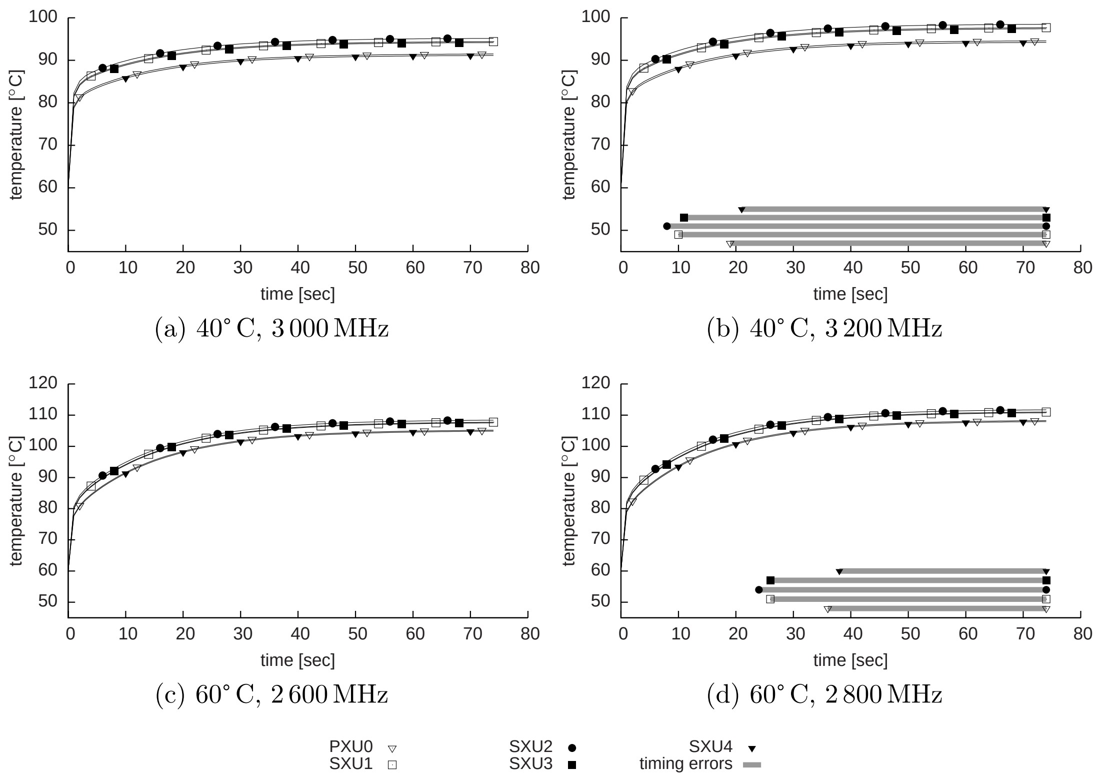


Figure 6.4: Temperature course in the simulation of the MPSoC without XCS at 20° C ambient temperature. Top left: at 3 200 MHz. Top right: at 3 400 MHz. Bottom: at 3 600 MHz.

3 200 MHz, 3 400 MHz, and 3 600 MHz. All diagrams depict the initial steep temperature rise as also found in Section 6.1. At 3 200 MHz, the diagram shows no timing errors. At 3 400 MHz, the inner cores SXU1, SXU2, and SXU3 show immediate timing errors for a short period of time, while the outer cores PXU0 and SXU4 do not show timing errors. At 3 600 MHz, all cores show immediate timing errors. For the inner cores, the time period in which timing errors occur is larger than for the outer cores.

The in-advance study at 20° C shows that an error-free operation is possible up to a frequency of 3 200 MHz. The initial overshoot of the core temperature over the longterm steady state temperature is because the heating of the cores is larger than the cooling capacity of the heat sink. The higher frequencies exhibit the different cooling properties of the cores. At 3 200 MHz, only the inner cores show timing errors, albeit only over a short period of time, while the outer cores do not show timing errors. At 3 600 MHz, all cores show timing errors, with the inner cores having timing errors over a longer period of time. Considering that potentially all cores exchange classifiers, sharing classifiers between the outer cores that show no timing errors at 3 200 MHz and the inner cores that do show timing errors, poses a difficulty to the XCS.

Figure 6.5 shows the temperature course of all cores for 40° C and 60° C ambient temperature. At 40° C, no core shows timing errors at 3 000 MHz (Figure 6.5a), while at 3 200 MHz (Figure 6.5b) and above (not depicted), all cores show delayed timing errors. Error-free operation at 40° C ambient temperature is thus possible up to 3 000 MHz. The delayed timing errors at 3 200 MHz pose an additional difficulty for the XCS, as their occurrence is influenced by the neighboring cores and, later, by the varying activities of



(a) 40° C, 3 000 MHz

(b) 40° C, 3 200 MHz

(c) 60° C, 2 600 MHz

(d) 60° C, 2 800 MHz

Figure 6.5: Temperature course in the simulation of the MPSoC without XCS at (top) 40° C and (bottom) 60° C ambient temperature and different frequencies.

the cores. For example, at SXU2, the timing errors start earlier than at SXU4, because SXU2 is additionally heated by its neighboring cores.

Figure 6.5c and 6.5d show the temperature course of all cores at 60° C ambient temperature and 2 600 MHz and 2 800 MHz, respectively. At 2 600 MHz, no core shows timing errors, while at 2 800 MHz and above (not depicted), all cores show delayed timing errors. Error-free operation at 60° C ambient temperature is thus possible up to 2 600 MHz. This fulfills the assumption of the reward function that an error-free operation is possible at any ambient temperature. The larger delay until a timing error occurs compared to 40° C ambient temperature is due to the lower frequency.

Table 6.1 summarizes the maximum frequency that is achievable in the simulation model without timing errors, for the various ambient temperatures.

6.2.2 Isolated, non-cooperating XCS

An analysis with isolated, non-cooperating XCS instances as in Section 6.1 shows the difficulty of the control problem of the MPSoC. As the XCS instances are isolated, there is no migration strategy and no topology to vary. First, the results for deleting excessive classifiers by prediction error ($\mathcal{D} = E$) are presented, followed by the results for deleting

Table 6.1: Maximum achievable frequency without timing errors in the MPSoC simulation model.

ambient temperature	maximum frequency
20° C	3 200 MHz
40° C	3 000 MHz
60° C	2 600 MHz

by fitness ($\mathcal{D} = F$). Both deletion strategies are analyzed for all don't-care probabilities. As mentioned previously, the results are averages of the application parts over three runs.

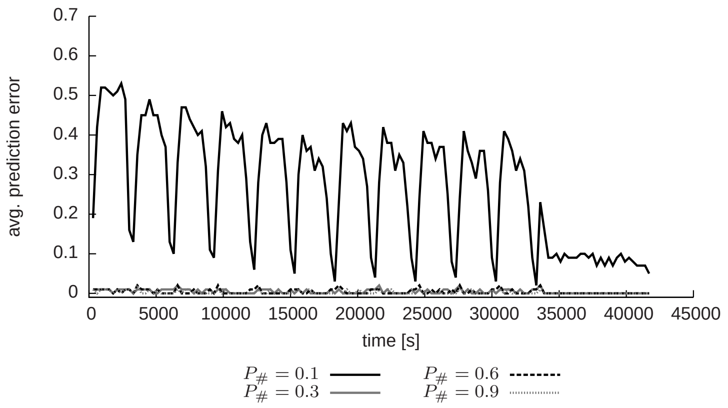
Deletion by prediction error Figure 6.6 shows the average prediction error and average success rate of the isolated, non-cooperating XCS when deleting by prediction error ($\mathcal{D} = E$) for different don't-care probabilities $P_{\#}$. For $P_{\#} = 0.1$, the average prediction error stays high and oscillates between about 0.1 and 0.4 along with the changing ambient temperature. For other values of $P_{\#}$, the average prediction error stays well below 0.05 but never stays at 0.0. For any value of $P_{\#}$, the average success rate quickly reaches values above 0.95, but never reaches constantly 1.0.

Figure 6.7 details the findings for the individual don't-care probabilities for the SXU2 core; other cores show comparable behavior. The two top lines show the core and ambient temperature, respectively. The triangles show the occurrence of immediate and delayed timing errors. The dots show the current frequency and voltage settings. With $P_{\#} = 0.1$, the XCS selects very low frequencies and voltage settings and shows very few delayed timing errors after about 6 000 seconds of simulated time. With other don't-care probabilities, the XCS selects a frequency of 3 400 MHz for low temperatures and 3 000 MHz for high temperatures; it always selects a high voltage. For $P_{\#} = 0.9$, the XCS shows fewer timing errors than for $P_{\#} = 0.3$ or $P_{\#} = 0.6$.

Deletion by fitness Figure 6.8 shows the average prediction error and average success rate of the isolated, non-cooperating XCS when deleting by fitness ($\mathcal{D} = F$) for different don't-care probabilities $P_{\#}$. While the ambient temperature changes, the average prediction error quickly drops below 0.05 but stays above 0.0. The average success rate increases above 0.95 but stays below 1.0, for any value of $P_{\#}$. For larger values of $P_{\#}$, the average prediction error drops more quickly and the average success rate increases faster. Only after the ambient temperature stays constant an average prediction error of 0.0 and an average success rate of 1.0 can be reached.

Figure 6.9 details the findings for the individual don't-care probabilities for the SXU2 core; other cores show comparable behavior. Except for $P_{\#} = 0.1$, the XCS selects a frequency of 3 200 MHz and 3 400 MHz at the highest voltage setting. There are many delayed timing errors, but hardly any immediate timing errors (except for $P_{\#} = 0.3$).

Summary The figures show that larger values for $P_{\#}$ are beneficial, but the isolated, non-cooperating XCS cannot reach an optimal setting when deleting classifiers by fitness. The average prediction error is smaller and the average success rate is larger for larger values



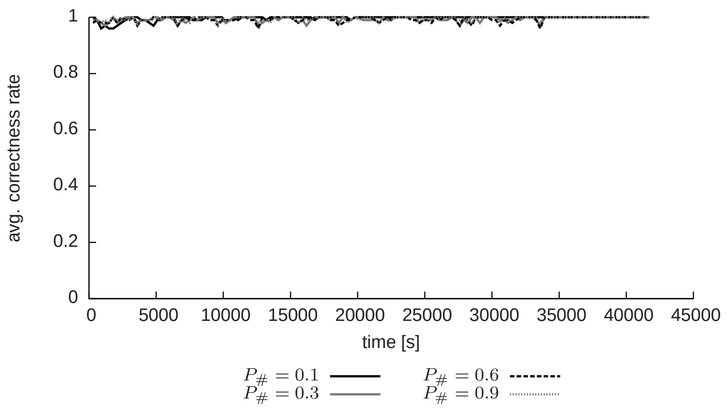
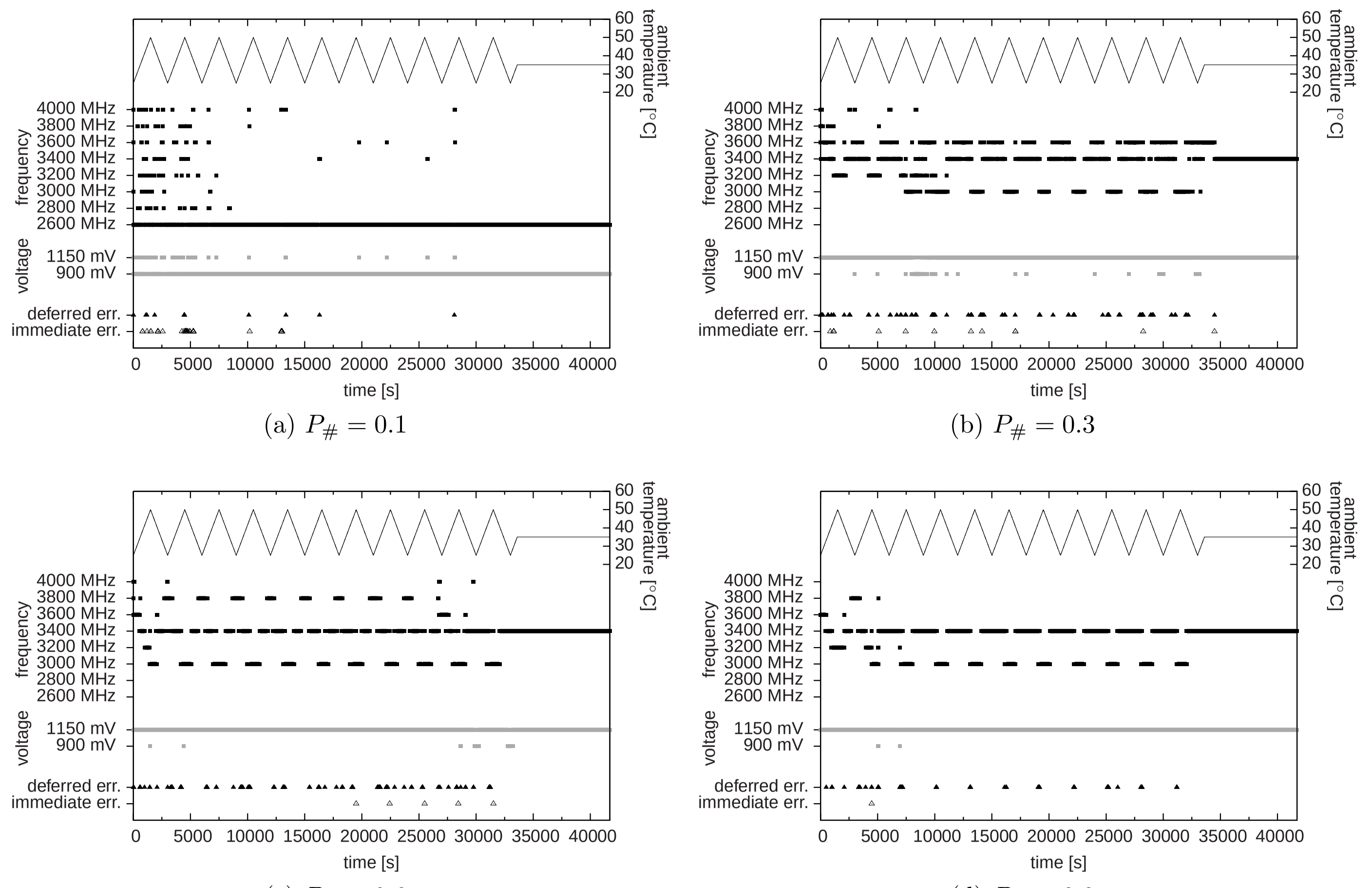


Figure 6.6: Average prediction error (top) and success rate (bottom) of the isolated, non-cooperating XCS when deleting by prediction error.

of $P_\#$. The selected frequencies are near the optimal frequencies found in the preliminary analysis for high activity and 40° C ambient temperature (the average temperature of the temperature cycles in the application part) and are reasonable for the reduced activity in the applied trace files, as the (mostly complete) absence of immediate timing errors also indicate. However, the XCS does not switch the frequency soon enough to avoid delayed timing errors.

The results when deleting by prediction error are slightly better than when deleting by fitness. However, the isolated, non-cooperating XCS cannot reach an optimal configuration under this setting, neither. Similar to when deleting by fitness, larger values for $P_\#$ are beneficial with lower average prediction errors and higher success rates. Yet, the XCS still does not switch the frequency soon enough to avoid delayed timing errors.

Overall, the results show that isolated, non-cooperating XCS cannot reach optimal settings, independent of the don't-care probabilities or the deletion strategy. This finding is in contrast to the results of Section 6.1, where isolated XCS instances could reach the optimal settings. The reason for this difference lies in the increased activity variability, higher transistor density (and thus power density), and higher temperature influence by neighboring cores: when the transistor density is halved or the activity is lowered to



(a) $P_{\#} = 0.1$

(b) $P_{\#} = 0.3$

(c) $P_{\#} = 0.6$

(d) $P_{\#} = 0.9$

Figure 6.7: Influence of the don't-care probability $P_{\#}$ in the application part with isolated, non-cooperating XCS when deleting by fitness ($\mathcal{D} = F$). The error-free time period at the end is because of the constant temperature.

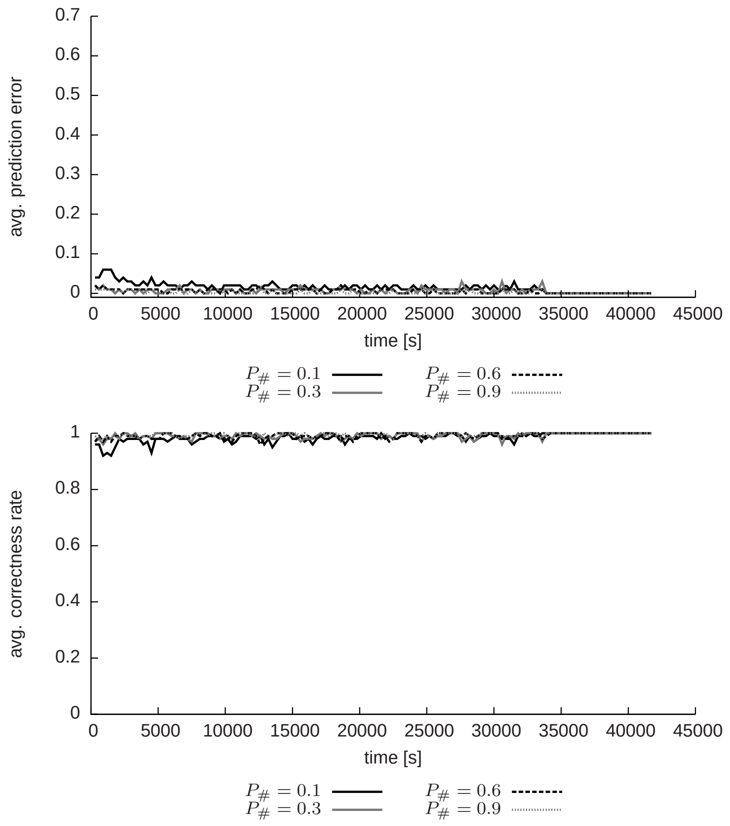


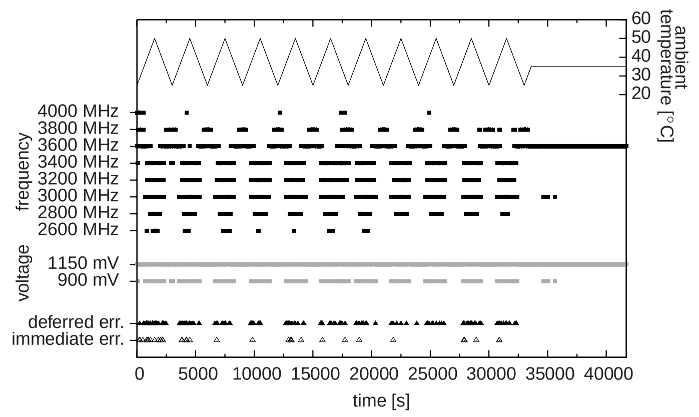
Figure 6.8: Average prediction error (top) and success rate (bottom) of the isolated, non-cooperating XCS when deleting by fitness ($\mathcal{D} = F$).

settings comparable to Section 6.1, the isolated XCS instances reach the optimal settings (not presented). The present MPSoC simulation model thus allows to show the benefits of distributed, cooperating XCS, if any.

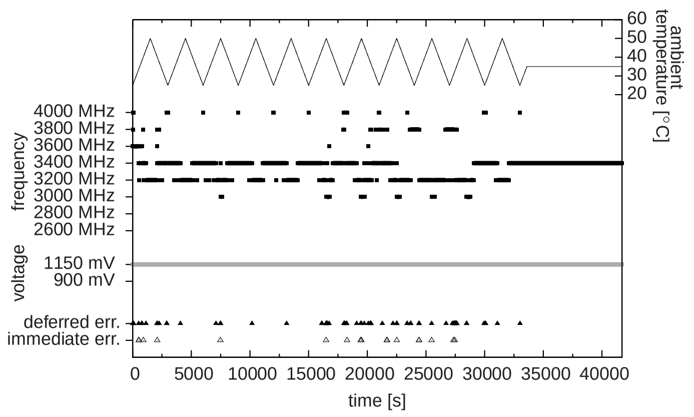
6.2.3 Distributed, cooperating XCS with emigrants selected at random

The first analysis of the distributed, cooperating XCS instances uses the simplest emigration strategy, namely emigration at random ($\mathcal{M} = R$). First, the results for deleting excessive classifiers by prediction error ($\mathcal{D} = E$) are presented, followed by the results for deleting by fitness ($\mathcal{D} = F$). Both deletion strategies are analyzed for all topologies and all don't-care probabilities. As mentioned previously, the results are averages of the application parts over three runs.

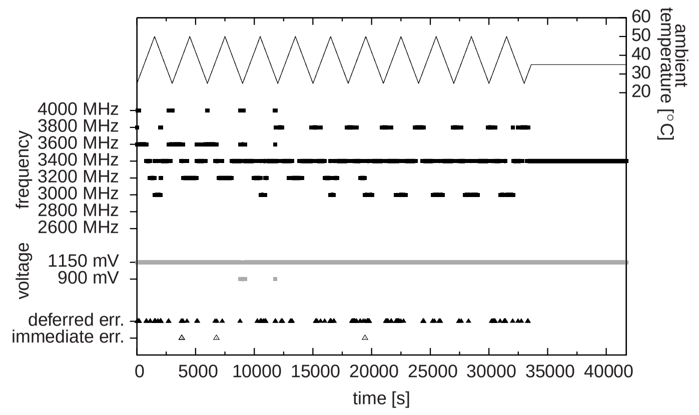
Deletion by prediction error Figure 6.10 shows the average prediction error and average success rate depending on the don't-care probability $P_{\#}$, averaged over all topologies. The average prediction error does not decrease and stays over about 0.6 for all don't-care probabilities except $P_{\#} = 0.9$. Although the average prediction error is lower for



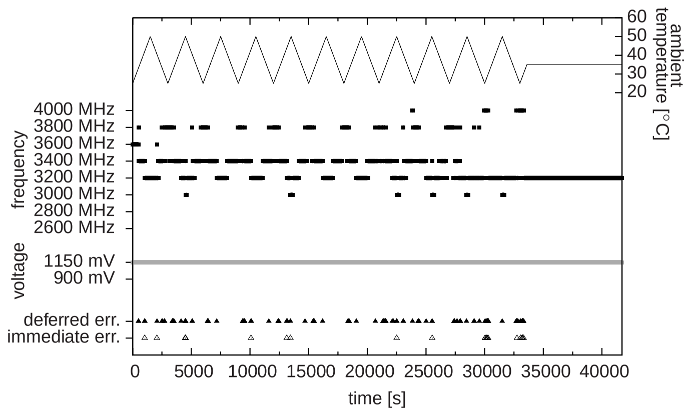
(a) $P_{\#} = 0.1$



(b) $P_{\#} = 0.3$



(c) $P_{\#} = 0.6$



(d) $P_{\#} = 0.9$

Figure 6.9: Influence of the don't-care probability $P_{\#}$ with isolated, non-cooperating XCS when deleting by fitness ($\mathcal{D} = F$). The error-free time period at the end is because of the constant temperature.

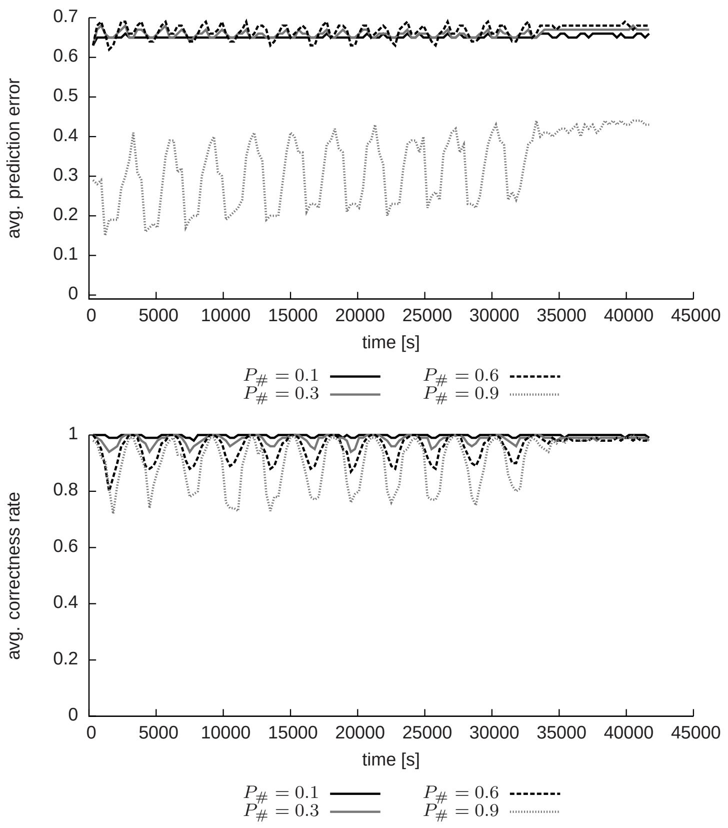


Figure 6.10: Average prediction error (top) and success rate (bottom) of the distributed XCS when emigrating at random and deleting by prediction error across all topologies, depending on $P_{\#}$.

$P_{\#} = 0.9$, it is still larger than 0.1 and increases over time. The average success rate stays above 0.95 only for $P_{\#} = 0.1$, but never reaches constant 1.0. For the other don't-care probabilities, $P_{\#} = 0.3$, $P_{\#} = 0.6$, and $P_{\#} = 0.9$, the average success rate oscillates synchronous to the temperature cycle, too, from 0.75, 0.85, and 0.95, respectively, to 1.0.

To further analyze the influence of the topology on the results, Figure 6.11 shows the average prediction error and average success rate depending on the topology for $P_{\#} = 0.9$, as it has the lowest prediction error. Other don't-care probabilities show similar behavior. For the unidirectional ring ($\mathcal{T} = U$), the average prediction error reaches constant 0.0 after less then 2 000 simulated seconds. However, for the bidirectional ring ($\mathcal{T} = B$) and the complete graph ($\mathcal{T} = C$), the average prediction error initially drops until about 6 000 simulated seconds, and increases thereafter, reaching values of 0.6 and above. The average success rate oscillates with amplitudes between 0.9 and 1.0 for the unidirectional ring ($\mathcal{T} = U$) and between 0.75 and 1.0 for the bidirectional ring ($\mathcal{T} = B$) and the complete graph ($\mathcal{T} = C$). It never reaches constant 1.0.

Figure 6.12 details the findings for the three topologies with $P_{\#} = 0.9$, exemplified at the core SXU2. Other cores show a similar behavior. With the unidirectional ring

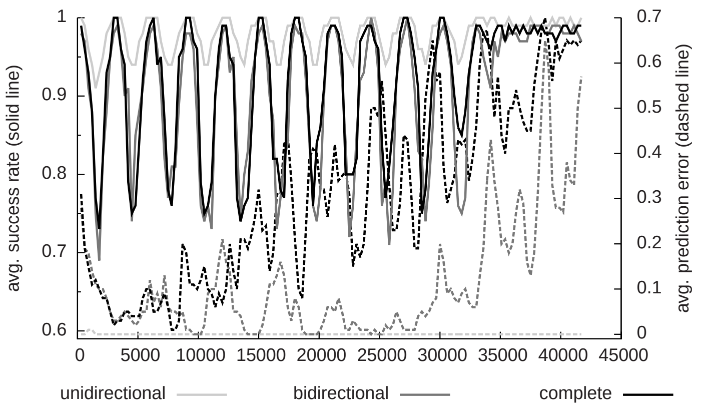


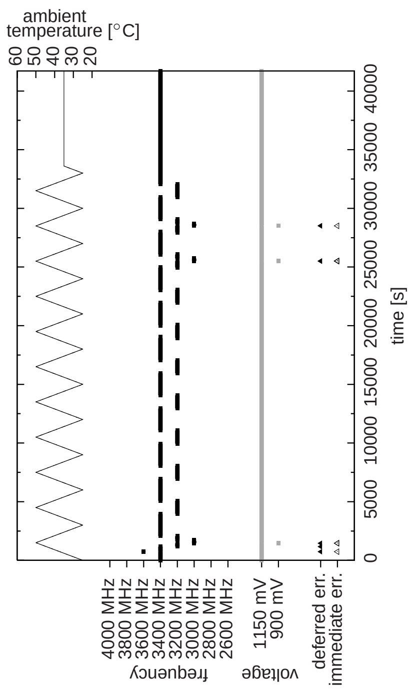
Figure 6.11: Average success rate (solid line, left y axis) and prediction error (dashed line, right y axis) of the distributed XCS when emigrating at random and deleting by prediction error for $P_{\#} = 0.9$, depending on topology.

($\mathcal{T} = U$), the XCS instance selects 3 400 MHz and 3 200 MHz as frequency, depending on the ambient temperature. After a grace period of less than 1 000 simulated seconds, almost no timing errors occur. With the bidirectional ring ($\mathcal{T} = B$) and the complete graph ($\mathcal{T} = C$), no pattern can be observed. The XCS seems to select the frequency and voltage at random and the rate of timing errors, both immediate and delayed, stays high. No learning can be observed.

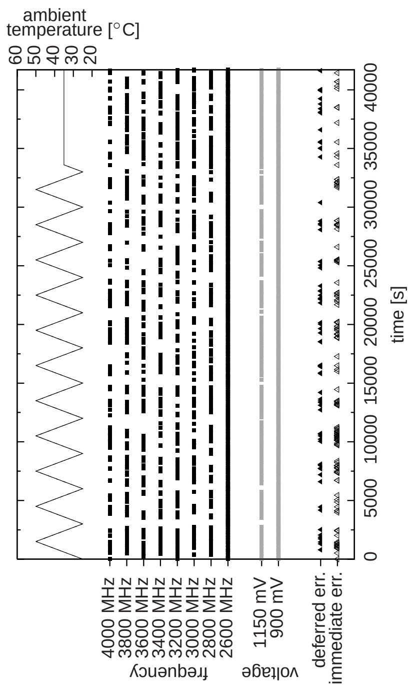
Deletion by fitness Figure 6.13 shows the average prediction error and average success rate depending on the don't-care probability $P_{\#}$, averaged over all topologies. The average prediction error oscillates between 0.05 and 0.2 for $P_{\#} = 0.1$, synchronous to the cycles of the ambient temperature. For $P_{\#} = 0.3$, the oscillation lies between 0.0 and 0.1. Only for $P_{\#} = 0.6$ and $P_{\#} = 0.9$, the average prediction error stays below 0.05. In no case does the average prediction error reach constant 0.0.

The average success rate also oscillates synchronous to the cycles of the ambient temperature, too. For $P_{\#} = 0.1$ and $P_{\#} = 0.3$, the oscillation lies between 0.85 and 1.0, while for $P_{\#} = 0.6$ and $P_{\#} = 0.9$, the oscillation lies between 0.95 and 1.0. In no case does the average success rate reach constant 1.0.

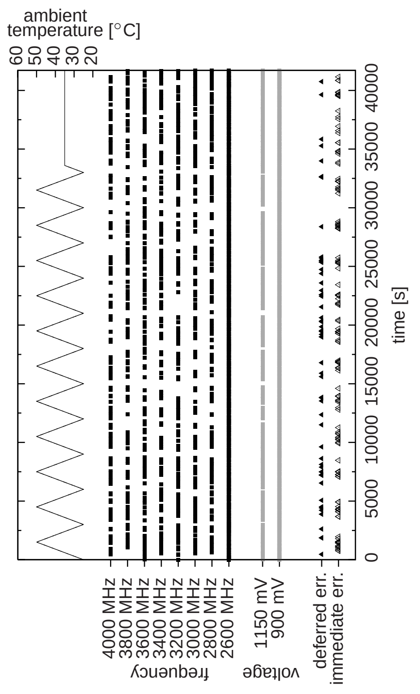
Summary Overall, when the XCS selects its emigrants at random ($\mathcal{M} = R$), hardly any learning can be observed, independent of the deletion strategy. The sole exception occurs with the unidirectional ring topology, deletion by prediction error and high don't-care probability ($\mathcal{T} = U, \mathcal{D} = E, P_{\#} = 0.9$). However, even at this setting, timing errors still occur.



(a) Unidirectional ring

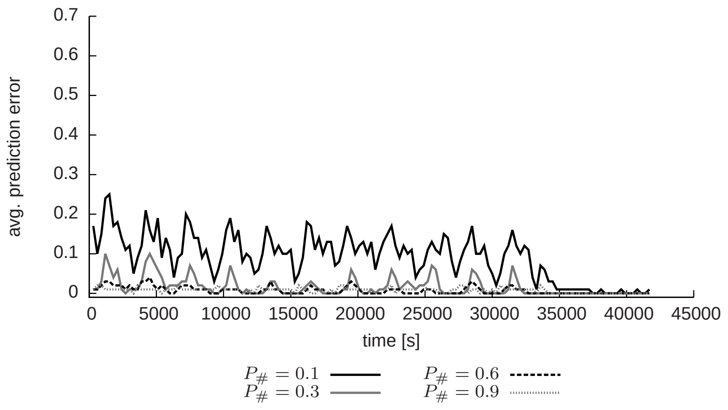


(b) Bidirectional ring



(c) Complete graph

Figure 6.12: Influence of the topology for distributed XCS with emigrants selected at random and deletion by prediction error for the core SXU2 with $P_{\#} = 0.9$.



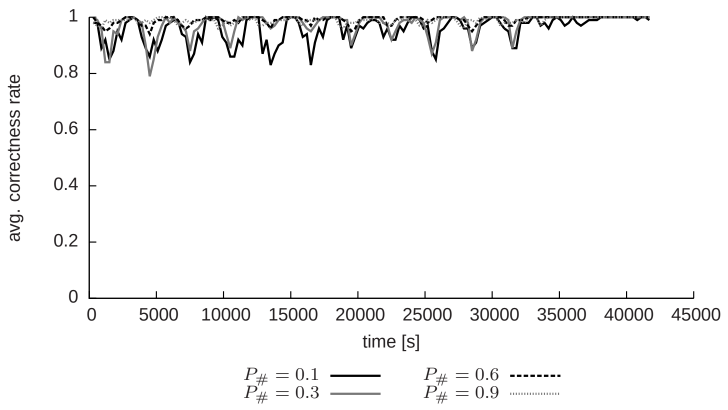


Figure 6.13: Average prediction error (top) and success rate (bottom) depending on $P_{\#}$ of the distributed, cooperating XCS when emigrating at random ($\mathcal{M} = R$) and deleting by fitness ($\mathcal{D} = F$), averaged over all topologies.

6.2.4 Distributed, cooperating XCS with emigrants selected by numerosity

The second analysis of the distributed, cooperating XCS instances uses the emigration strategy by numerosity ($\mathcal{M} = N$), that is, the XCS emigrates classifiers with higher numerosity first. The expectation is that classifiers that are numerous in the local population are beneficial in the other populations, too. First, the results for deleting excessive classifiers by fitness ($\mathcal{D} = F$) are presented, followed by the results for deleting by prediction error ($\mathcal{D} = E$). Both deletion strategies are analyzed for all topologies and all don't-care probabilities. As mentioned previously, the results are averages of the application parts over three runs.

Deletion by fitness Figure 6.14 shows the average prediction error and average success rate depending on the don't-care probabilities, averaged over all topologies. The average prediction error is below 0.05 for all don't-care probabilities, with the average prediction error for $P_{\#} = 0.1$ staying above the average prediction error of the other don't-care probabilities. However, the average prediction error never reaches 0.0. The average

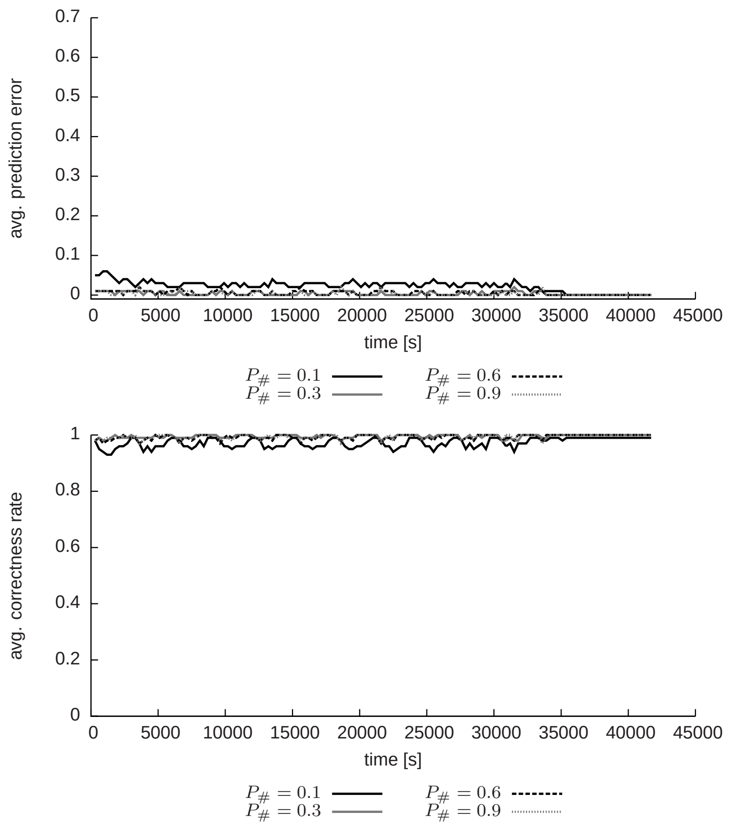


Figure 6.14: Average prediction error (top) and success rate (bottom) of the distributed XCS when emigrating by numerosity and deleting by fitness across all topologies, depending on $P_\#$.

success rate stays above 0.95 for all don't-care probabilities, with the average success rate for $P_\# = 0.1$ staying below the average success rate of the other don't-care probabilities. However, the average success rate never reaches 0.0.

The finding is similar to the previous findings: the higher the don't-care probability, the better the learning. Also, $P_\# = 0.1$ seems not as beneficial as other don't-care probabilities.

To further analyze the influence of the topology on the results, Figure 6.15 shows the average prediction error and average success rate averaged over the two top-performing don't-care probabilities, $P_\# = 0.6$ and $P_\# = 0.9$ for each topology. The average prediction errors are well below 0.05 and are hardly distinguishable between the different topologies, but never reach constant 0.0. The average success rate is well above 0.95 for all topologies. The average success rate of the complete graph topology ($\mathcal{T} = C$) is higher than the average success rate of the other topologies most of the time.

As the complete graph showed a higher success rate than the other topologies, Figure 6.16 details the findings for the don't-care probabilities for the SXU2 core with complete graph topology ($\mathcal{T} = C$). Other cores show comparable behavior. For $P_\# = 0.1$

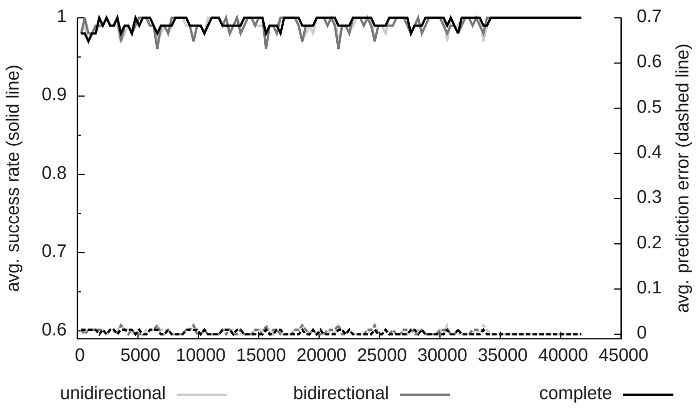


Figure 6.15: Average success rate (solid line, left y axis) and average prediction error (dashed line, right y axis) of the distributed XCS when emigrating by numerosity and deleting by fitness, averaged over $P_{\#} = 0.6$ and $P_{\#} = 0.9$, depending on the topology.

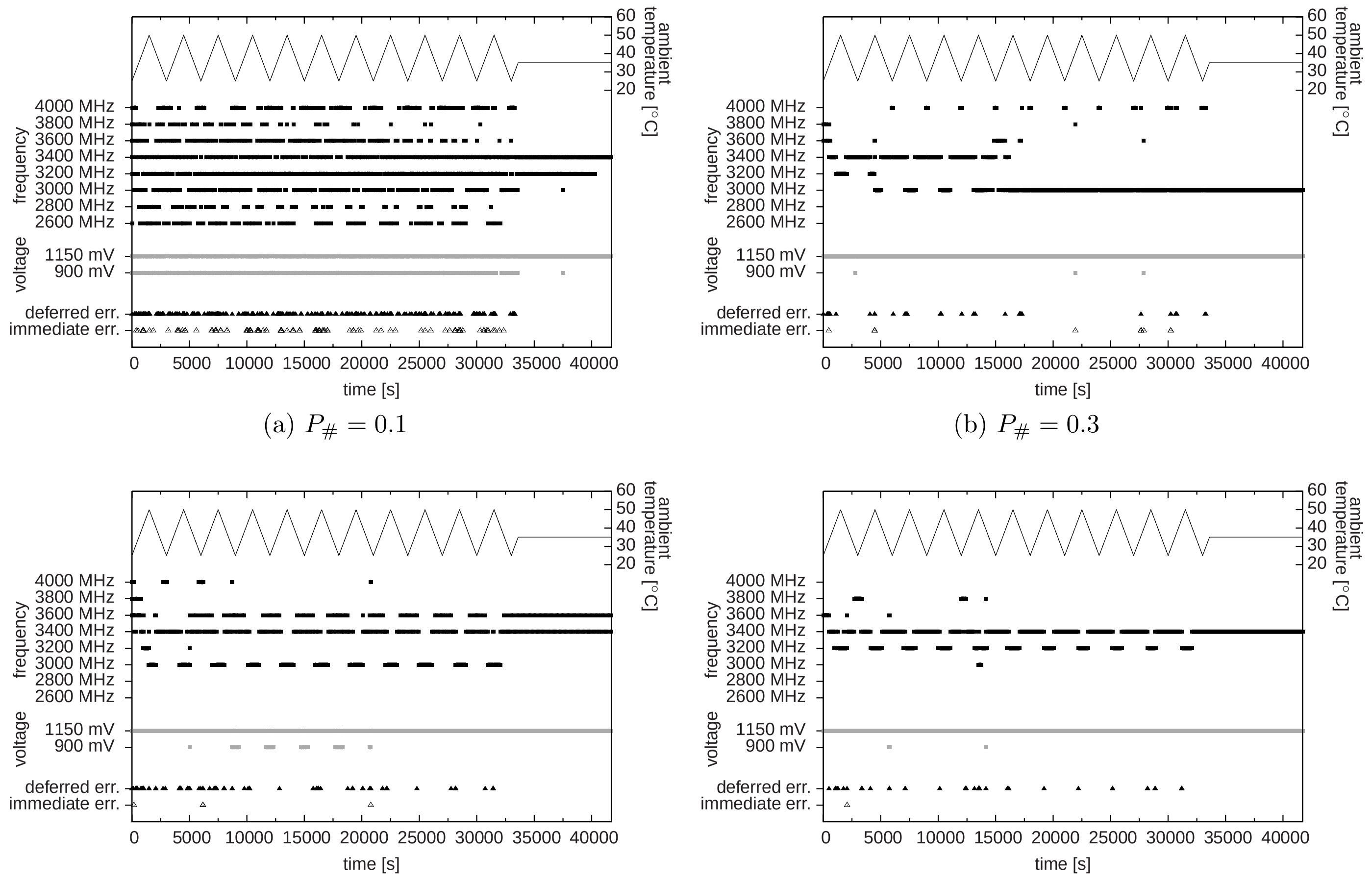
hardly any learning can be observed: the XCS selects the frequencies almost equally likely—only towards the end the XCS avoids the frequencies 3 600 MHz and 3 800 MHz and prefers higher voltages. The are numerous timing errors. For $P_{\#} = 0.3$, the XCS settles from 3 400 MHz to 3 000 MHz and stays at selecting high voltage. For $P_{\#} = 0.6$, the XCS selects 3 400 MHz and 3 600 MHz with high voltage most of the time. For $P_{\#} = 0.9$, the XCS selects 3 400 MHz most of the time and always high voltage. For the don't-care probabilities other than $P_{\#} = 0.1$, there are several timing errors, but less than for $P_{\#} = 0.1$.

In any case, the XCS does not achieve error free operation.

Deletion by prediction error Figure 6.17 shows the average prediction error and average success rate depending on the don't-care probability $P_{\#}$, averaged over all topologies. The average prediction errors for $P_{\#} = 0.1$ and $P_{\#} = 0.3$ stay almost constant slightly above 0.6 and 0.25, respectively, while the average prediction error for $P_{\#} = 0.6$ and $P_{\#} = 0.9$ reach constant 0.0 quickly. The average success rate for $P_{\#} = 0.1$ and $P_{\#} = 0.3$ stay above 0.95. Only the average success rate for $P_{\#} = 0.6$ and $P_{\#} = 0.9$ reach constant 1.0, with the average success rate for $P_{\#} = 0.6$ showing sporadic small drops.

To further analyze the influence of the topology on the results, Figure 6.18 shows the average prediction error and average success rate depending on the topology, averaged for $P_{\#} = 0.6$ and $P_{\#} = 0.9$, the don't-care probabilities with the lowest average prediction error. For all topologies, the average prediction error reaches constant 0.0 quickly, with sporadic bumps when using the unidirectional ring. Similarly, for all topologies the average success rate reaches constant 1.0 quickly, with sporadic drops when using the unidirectional ring.

Figure 6.19 details the findings for the don't-care probabilities for the SXU2 core with complete graph topology; other cores show similar behavior. For $P_{\#} = 0.1$, the XCS instance seems to select the frequencies at random; no learning can be observed and



(a) $P_\# = 0.1$

(b) $P_\# = 0.3$

(c) $P_\# = 0.6$

(d) $P_\# = 0.9$

Figure 6.16: Influence of the don't-care probability $P_\#$ with distributed XCS emigrating by numerosity and deleting by fitness, exemplified at the core SXU2 with complete graph topology.

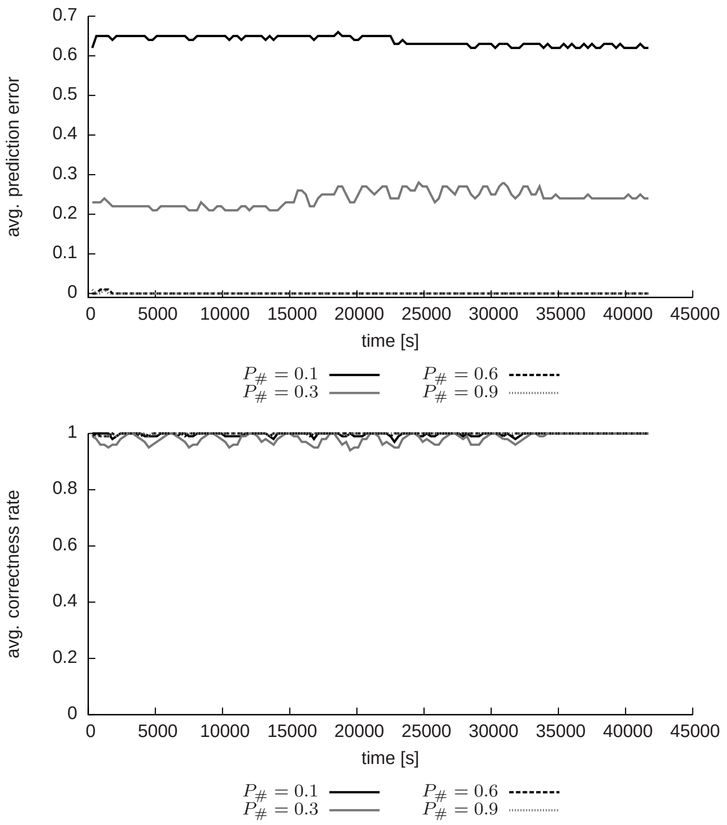


Figure 6.17: Average prediction error (top) and success rate (bottom) of the distributed XCS when emigrating by numerosity and deleting by prediction error across all topologies, depending on $P_{\#}$.

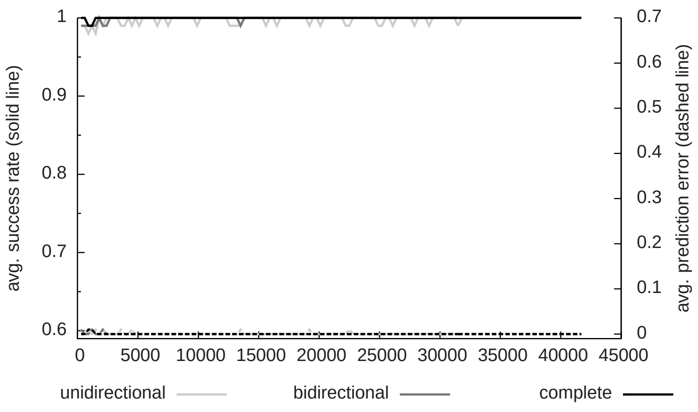


Figure 6.18: Average success rate (solid line, left y axis) and prediction error (dashed line, right y axis) of the distributed XCS when emigrating by numerosity and deleting by prediction error, averaged over $P_{\#} = 0.6$ and $P_{\#} = 0.9$, depending on topology.

there are numerous timing errors. For $P_{\#} = 0.3$, there are no timing errors, but the XCS instance constantly selects 3 000 MHz and high voltage, showing no self-adaptation to the changing ambient temperature. After a short grace period, there are no timing errors for $P_{\#} = 0.6$ and $P_{\#} = 0.9$ and the XCS instance toggles between 3 000 MHz and 3 400 MHz with high voltage, depending on the ambient temperature.

Summary When the XCS selects its emigrants by numerosity ($\mathcal{M} = N$), it is better able to avoid timing errors, in particular when deleting by prediction error. However, it selects frequencies that are too low when the ambient temperature peaks.

6.2.5 Distributed, cooperating XCS with emigrants selected by fitness

The third analysis of the distributed, cooperating XCS instances uses the emigration strategy by fitness ($\mathcal{M} = F$), that is, the XCS emigrates classifiers with higher fitness first. The expectation is that highly fit classifiers are beneficial in other populations, too, maybe after some small adjustment. The results for deleting excessive classifiers by prediction error ($\mathcal{D} = E$) are presented, analyzed for all topologies and all don't-care probabilities. The results for deleting excessive classifiers by fitness do not show an improvement (data not shown). As mentioned previously, the results are averages of the application parts over three runs.

Deletion by prediction error Figure 6.20 shows the average prediction error and average success rate depending on the don't-care probability $P_{\#}$, averaged over all topologies. Except for $P_{\#} = 0.1$, the average prediction error stays constantly at 0.0. The average prediction error for $P_{\#} = 0.1$ starts at 0.4, increases to 0.6 and oscillates. The average success rate stays constantly at 1.0, except for a small grace period, which is 9 000 simulated seconds for $P_{\#} = 0.1$ and less than 2 000 simulated seconds for the other don't-care probabilities.

To further analyze the influence of the topology on the results, Figure 6.18 shows the average prediction error and average success rate depending on the topology, averaged over $P_{\#} = 0.6$ and $P_{\#} = 0.9$. Except for a small bump at the beginning, the average prediction error stays constantly at 0.0 for all topologies. Similarly, the average success rate stays constantly at 1.0 for the bidirectional ring ($\mathcal{T} = B$) and complete graph ($\mathcal{T} = C$) topologies, except for a small drop at the beginning. The average success rate for the unidirectional ring ($\mathcal{T} = U$) is also 1.0 most of the time, with sporadic small drops.

Figure 6.21 details the findings for the SXU2 core with complete graph topology; other cores show comparable behavior. For $P_{\#} = 0.1$, there are several timing errors and the XCS does not seem to settle to a certain frequency. For $P_{\#} = 0.3$, the XCS constantly selects 3 000 MHz at high voltage, which ensures no timing errors, but also does not show any self-adaptation to the ambient temperature. After a short grace period, there are no timing errors for $P_{\#} = 0.6$ and $P_{\#} = 0.9$. The XCS instance repeatedly cycles between 3 000 MHz, 3 200 MHz, and 3 400 MHz for $P_{\#} = 0.6$ and between 3 400 MHz and

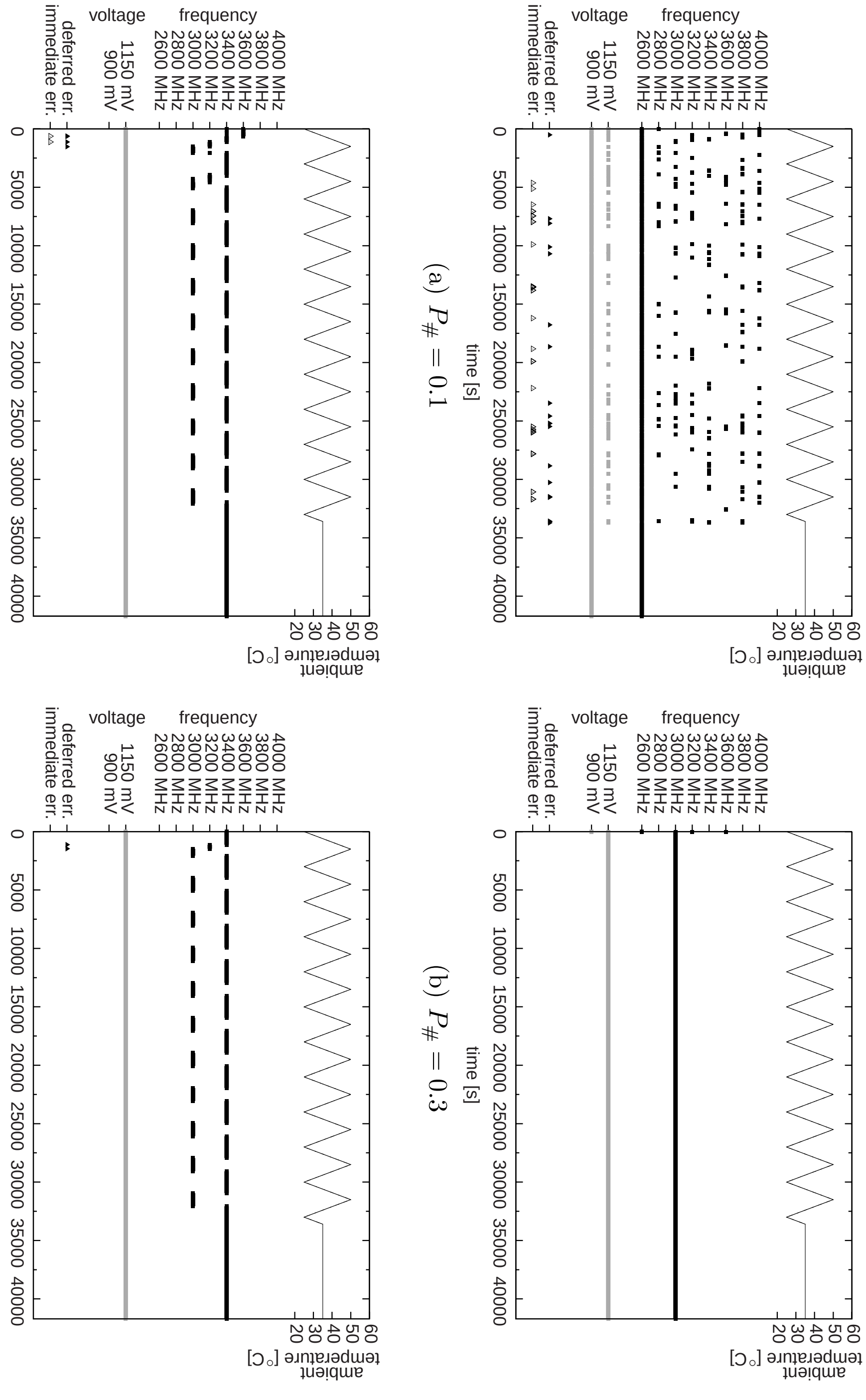
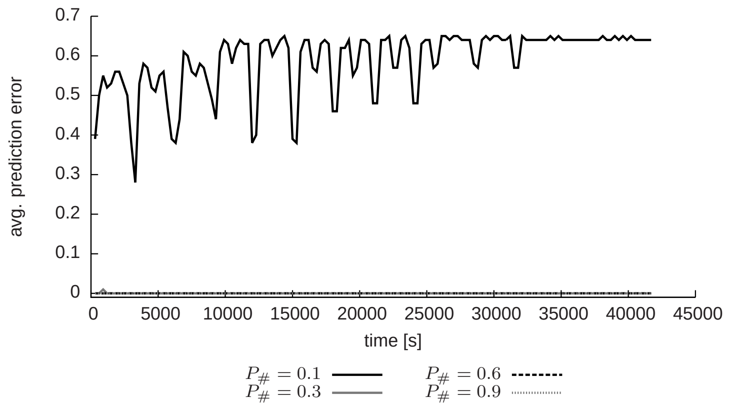


Figure 6.19: Influence of the don't-care probability $P_\#$ with distributed XCS emigrating by numerosity and deleting by prediction error, exemplified at the core SXU2 with complete graph topology.



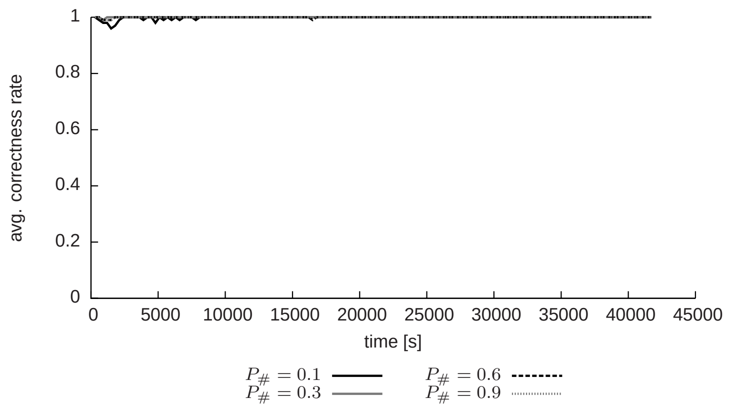


Figure 6.20: Average prediction error (top) and success rate (bottom) of the distributed XCS when emigrating by fitness and deleting by prediction error across all topologies, depending on $P_{\#}$.

3 000 MHz for $P_{\#} = 0.9$, depending on the ambient temperature, each with high voltage most of the time.

Summary When the XCS selects its emigrants by fitness and deletes by prediction error ($\mathcal{M} = F$), it can avoid almost any timing error. It selects appropriately high frequencies, in particular for $P_{\#} = 0.9$. For $P_{\#} = 0.6$, the selected frequencies are low enough to avoid timing errors, but not as high as possible.

6.2.6 Distributed, cooperating XCS adapting to changing environments

To show the self-adaptivity of the cooperating XCS to environments that differ from the environment of the previous analysis, two additional application parts are analyzed. In the first additional application part, the ambient temperature is kept constant at 45° C, a usual ambient temperature for chip operation. In the second additional application part, the activity of the processes running on the cores is increased and kept constant at 20%. The analysis are carried out with the classifiers of the best performing XCS configuration,

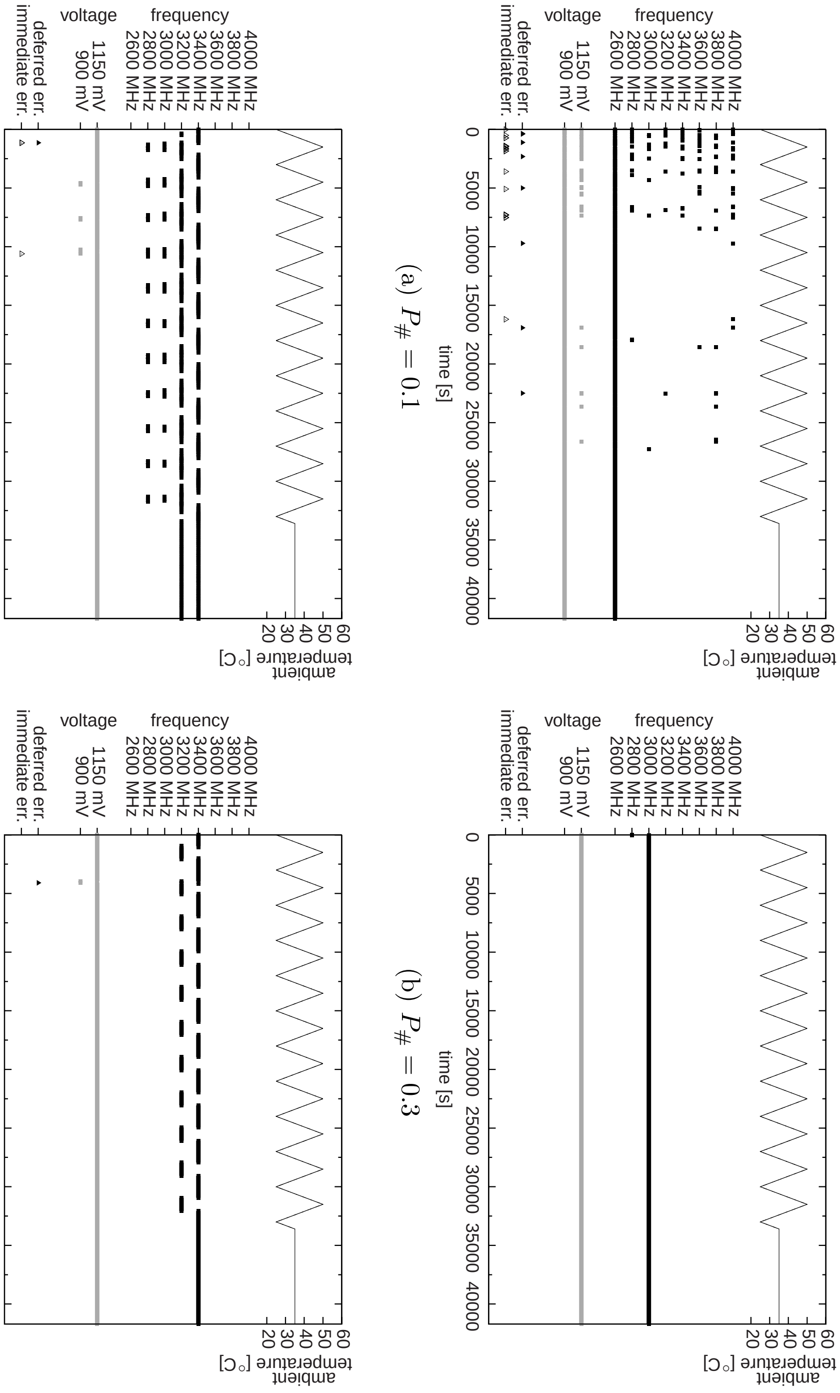


Figure 6.21: Influence of the don't-care probability $P_\#$ with distributed XCS emigrating by fitness and deleting by prediction erorr, exemplified at the core SXU2 with complete graph topology.

namely the XCS instance with emigrant selection by fitness, complete graph topology, and deletion by prediction error ($\mathcal{M} = F, \mathcal{T} = C, \mathcal{D} = E$).

Figure 6.22 shows the results with constant ambient temperature for every core and varying activity. The same activity trace file is used as previously. SXU2 shows a timing error at the very beginning and reduces its frequency from 3 400 MHz to 3 000 MHz. The other cores show no timing errors and reduce their frequency from 3 400 MHz to 3 200 MHz.

SXU2 is the center core and thus heated the most; hence, it shows the timing error before the other cores do and has to reduce its frequency lower than the other cores. Compared to the preliminary analysis of the simulation model in Section 6.2.1, where at 40° C ambient temperature and high activity the maximum possible frequency is 3 000 MHz, the frequency choice of 3 000 MHz for SXU2 and 3 200 MHz for the other cores is sensible.

Figure 6.23 shows the results with constant high activity and varying ambient temperature around 50° C. For a grace period of about 50 simulated seconds, all cores show timing errors. The XCS instances of all cores reduce the frequency to 2 800 MHz within that grace period, after which no timing errors occur. Considering the preliminary analysis, which shows the highest possible frequency for error-free operation to be 3 000 MHz at 40° C and 2 600 MHz at 60° C, the frequency choice of 2 800 MHz is sensible.

Summary Overall, with classifiers from the best strategy, the distributed XCS adapts to new environmental conditions quickly . Both an increased workload of the CPU cores and a change in the conduct of the ambient temperature can be coped with. Timing errors occur only in the beginning of the experiments, when the situation is new, and are handled by XCS correctly by decreasing frequency.

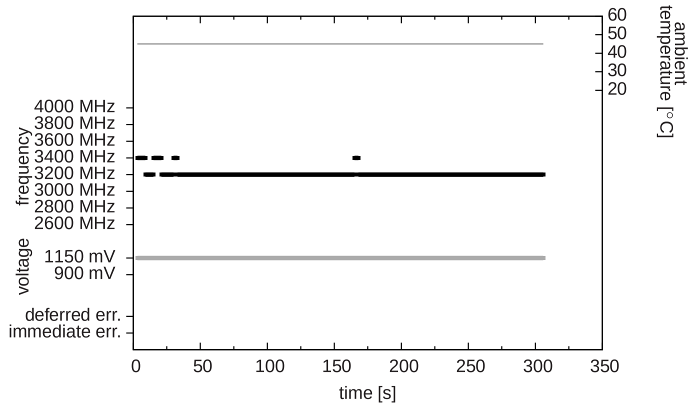
6.3 Solving the combinatorial operator-allocation problem

This section presents the results on applying the proposed design methodology to solve the operator-allocation problems. The results are published in Bernauer et al. (2009). Repeated simulations show similar patterns like presented in the following subsections and lead to the same conclusions. Each complete set of simulation experiments took about one hour on current hardware, parallelized over eight CPU cores.

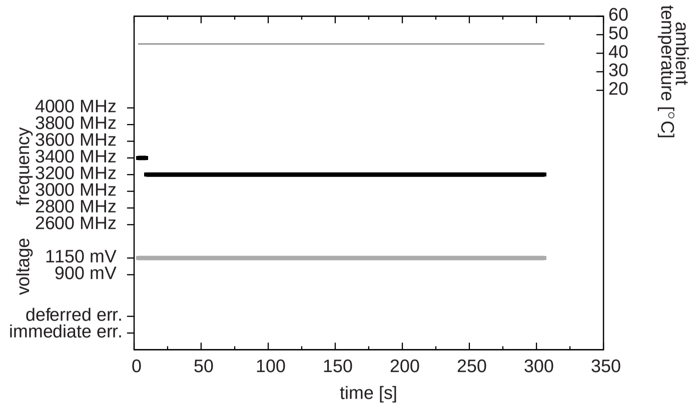
6.3.1 Basic operator-allocation problem

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 operators in the basic operator-allocation problem. The correctness rate that the XCS reaches is evaluated to determine its basic performance.

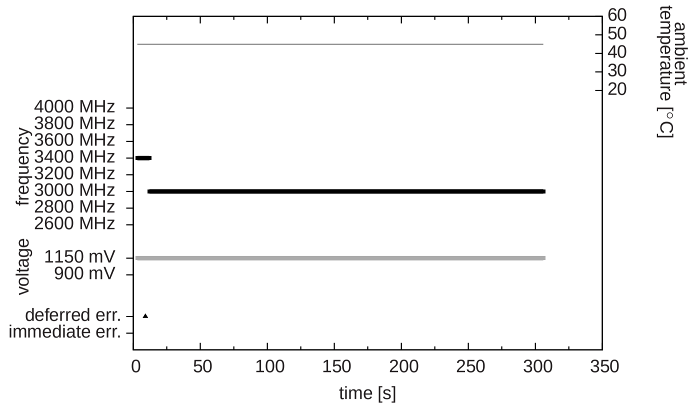
The results for the $(10, 3)$ and $(10, 5)$ operator-allocation problems are shown in Figure 6.24. The charts show the number of learning steps on the x axis, the correctness rate on the left y axis (labeled '%') and the number of classifiers on the right y axis



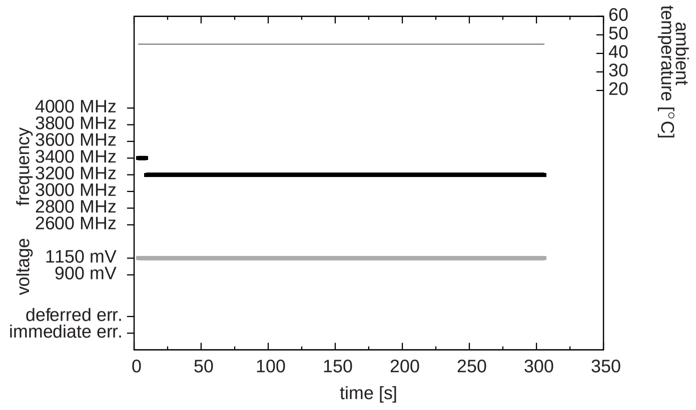
(a) PXU0



(b) SXU1

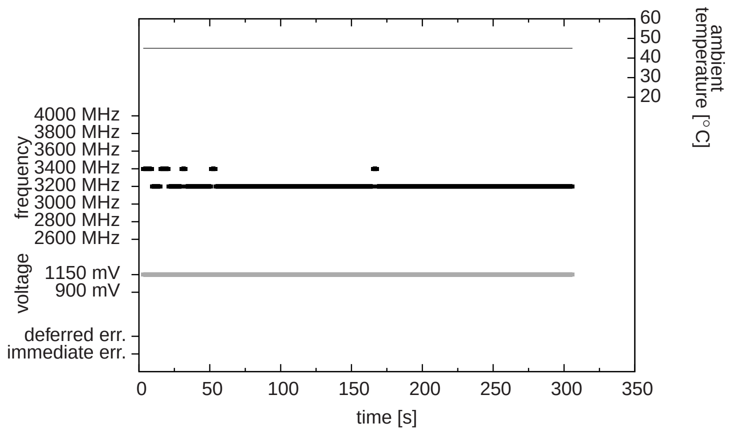


(c) SXU2



(d) SXU3

Figure 6.22: Additional third part with constant ambient temperature of 45° C. *Continued on next page.*



(e) SXU4

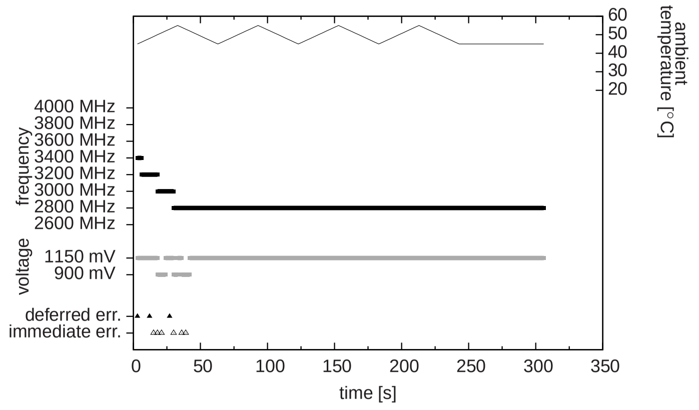
Figure 6.22: *Continued from previous page.* Core SXU4 in the additional third part with constant ambient temperature of 45° C.

(labeled 'classifiers'). Figure 6.24a 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 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 4.4). Figure 6.24b shows a similar pattern for the $(10, 5)$ operator-allocation problem. 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).

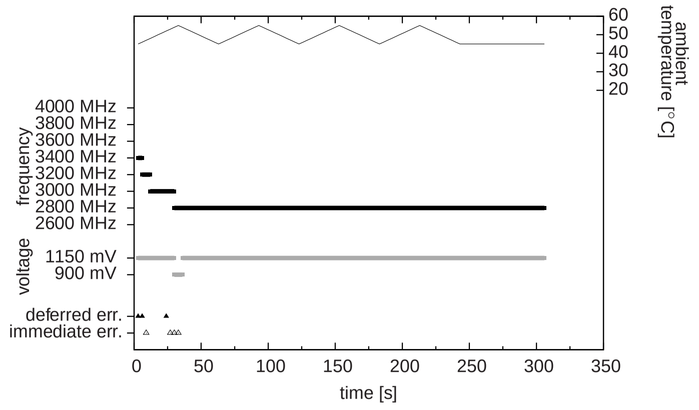
The population size initially increases as the XCS randomly creates classifiers to cover large parts of the problem space. Afterwards, the XCS combines similar classifiers using the wild card symbol. The population size is about 1 800 for the $(10, 3)$ allocation problem and about 2 100 for the $(10, 5)$ allocation problem when the correctness rate does not improve further.

Next, the experiments for all combinations of c and t with $1 \leq t \leq c \leq 10$ are run. The contour plot in Figure 6.25 depicts the correctness rate of the XCS at the end of the learning process. The learning process ends 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 operators (labeled 'operators') and the y axis shows the number of operators to be allocated (labeled 'n_{allocate}'). The contour plot contains an isoline for every five percentage points.

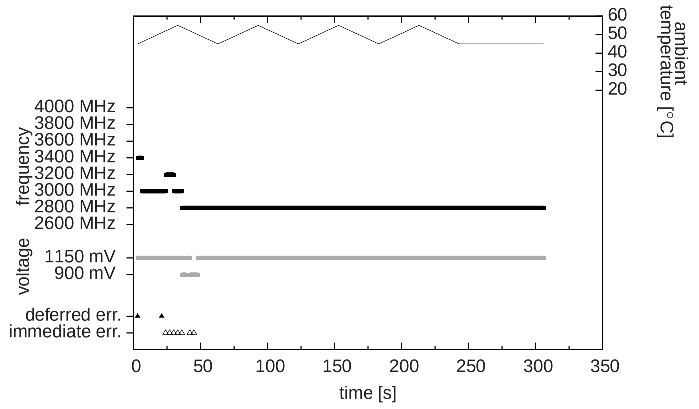
The XCS reaches a correctness rate of 90% or more for the simple problem configurations where the XCS has to allocate either only a few operators (white area near the x axis) or almost all operators (white area below the main diagonal). In between, the number of



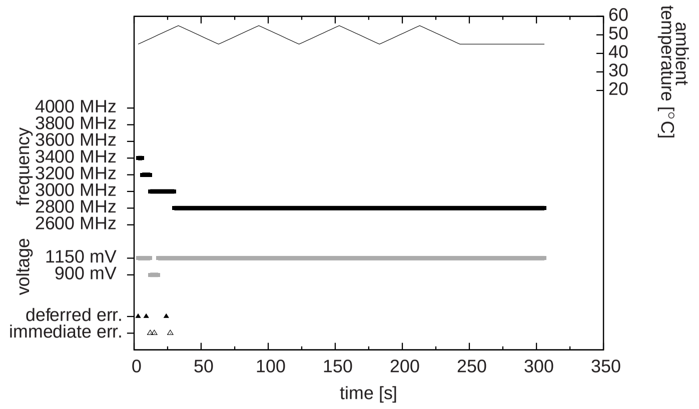
(a) PXU0



(b) SXU1

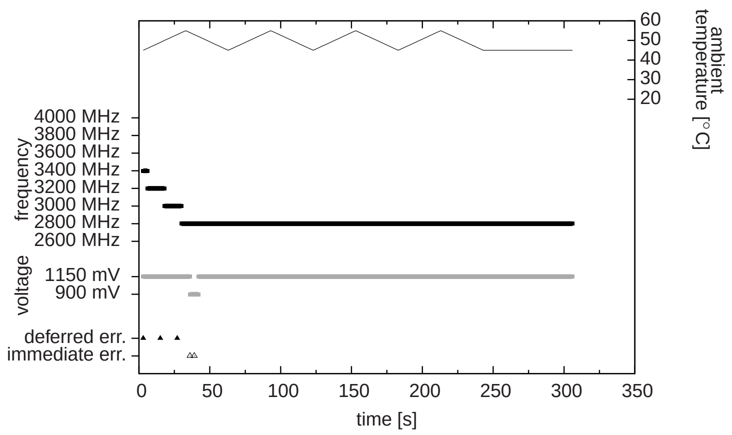


(c) SXU2



(d) SXU3

Figure 6.23: Additional third part with maximum activity and varying ambient temperature. *Continued on next page.*



(e) SXU4

Figure 6.23: *Continued from previous page.* Core SXU4 in the additional third part with maximum activity and varying ambient temperature.

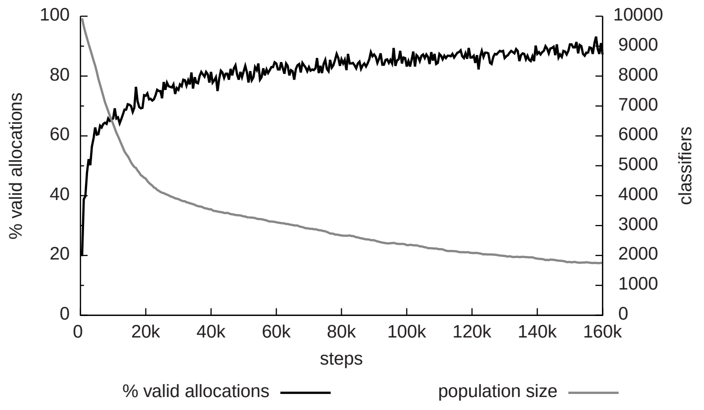
possible allocations is large, which means that the number of possible actions is large and the XCS has a harder time to figure out valid actions. However, the correctness rate reaches 68% (for the $(10, 6)$ operator-allocation problem) or more. Compared to Figure 4.4, the correctness rate of the XCS 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 $(10, 5)$ allocation problem, the XCS has a correctness rate of 71% whereas the random-allocation evaluator has a correctness rate of only 3%.

6.3.2 Self-adaptation to failing operators

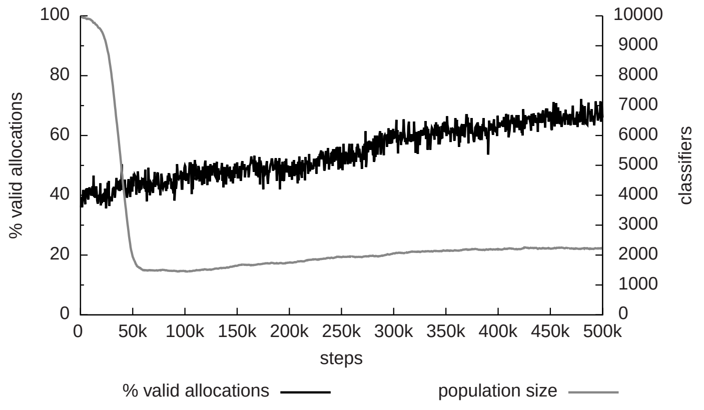
Next, the self-adaptation capabilities of XCS at run-time are analyzed. The basic performance at run-time is assumed to be the same as presented in Section 6.3.1.

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, the run-time behavior of the XCS is simulated as well as whether the XCS can tolerate the total failure of one ore more operators although the XCS has not had the opportunity to learn actions for these events at design time.

Figure 6.26 shows the correctness rates of the XCS and the LUT for the $(10, 3)$ and $(10, 5)$ operator-allocation problem, respectively, during operator failure. The arrows mark when a random operator fails. For the XCS, the result of a single run and a sequence of operator failures is depicted; other runs with different sequences of operator failures 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 6.26a, the first operator 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



(a) $(10, 3)$ allocation problem.



(b) $(10, 5)$ allocation problem.

Figure 6.24: Design-time learning of the XCS.

operator fails until three available operators are left over (at 1.7 million learning steps)[1]. If more operators failed, the system would become unusable in the intended sense. The figure shows that the correctness rate for the XCS drops from about 90% to 80% when half of the ten operators have total failures. After that, the correctness rate increases again to the usual level of 90%. In Figure 6.26b, the first operator is instructed to fail after 1.4 million learning steps and after that, every 200 000 steps another operator fails until five operators are left over (at 2.0 million learning steps). The figure shows that the correctness rate initially stays at the usual 70% level and increases when only six functioning operators are left over. The correctness rate for the LUT drops monotonically in both cases, with the first drop reaching less than 50%, as operator failures have not been considered during its design.

[1] The number of learning steps between operator failures does not have a significant effect on the results; it is chosen to be large enough so that the system can encounter all possible situations and thus meaningful correctness rates between operator failures can be acquired.

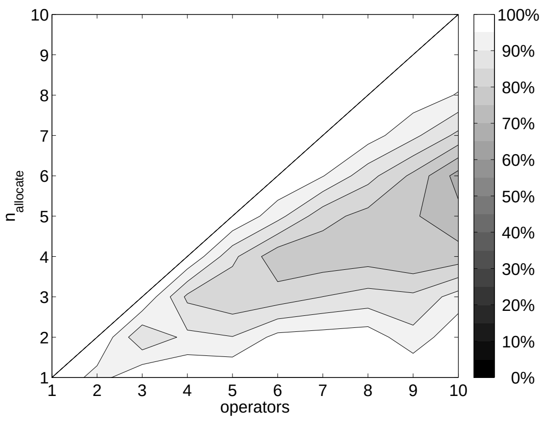
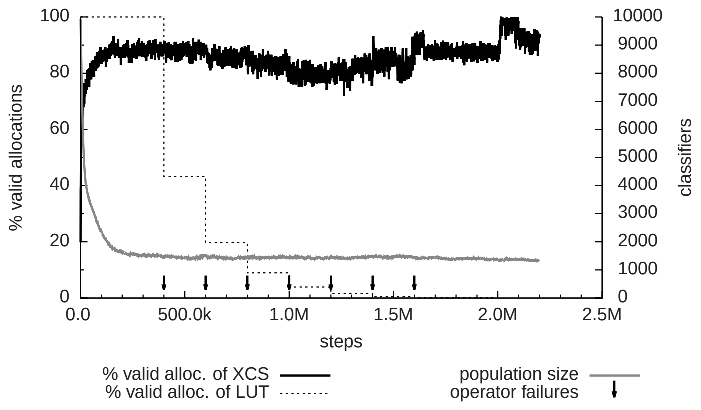
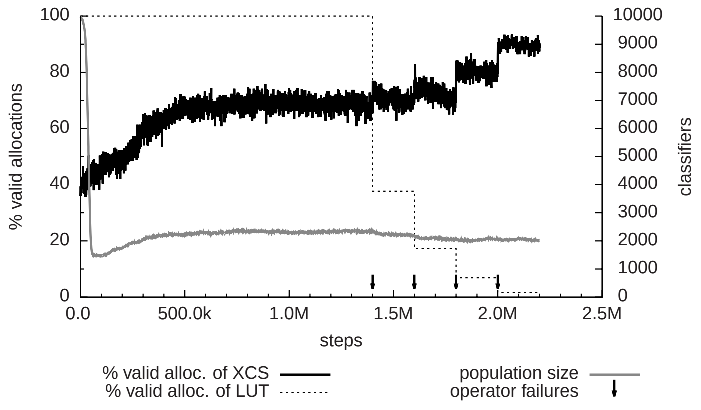


Figure 6.25: Probability of a valid allocation by the XCS in the basic operator-allocation problem.

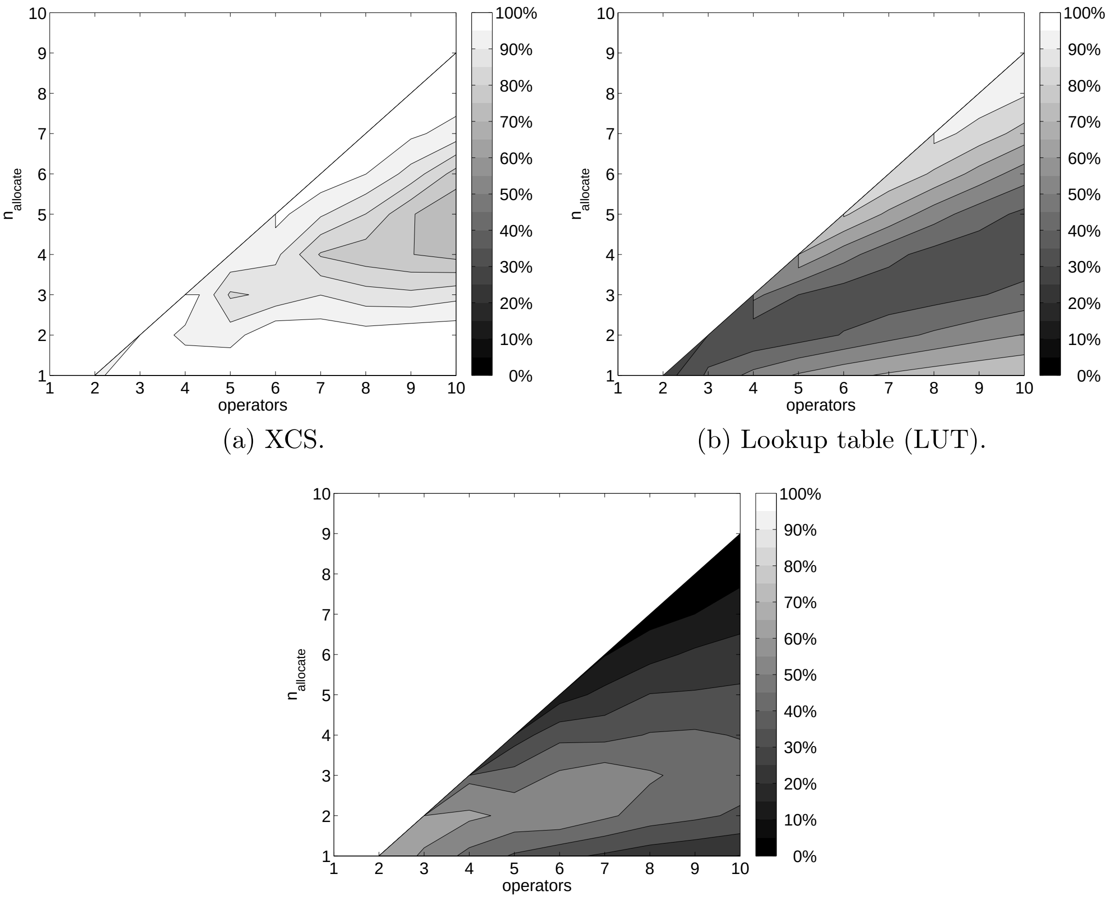


(a) $(10, 3)$ allocation problem.



(b) $(10, 5)$ allocation problem.

Figure 6.26: Self-adaptation to failing operators of the XCS at run time.



(a) XCS.

(b) Lookup table (LUT).

(c) Difference between XCS and LUT in percentage points.

Figure 6.27: Probability of a valid allocation after one operator has failed.

Figure 6.27a depicts the correctness rate of the XCS at 100 000 steps after the first operator has failed for all combinations of c and t with $1 \leq t \leq c \leq 10$. The figure shows that the correctness rate of the XCS after one operator has failed is at about 90% when either a few operators (near the x axis) or almost all operators (below the main diagonal) have to be allocated. In between, the correctness rate reaches almost 70% (for the $(10, 5)$ allocation problem) or more. Compared to Figure 6.25, the XCS is able to maintain the correctness rate of the basic operator-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 6.27b, which is an analytical result. The correctness rate of the LUT decreases significantly after one operator has failed. While for the basic operator-allocation problem the correctness rate of the LUT is 100% for all combinations by design, the correctness rate after one operator has failed drops to as low as 25%, as operator failures had not been considered explicitly during the design of the LUT. For the $(10, 3)$ and $(10, 5)$ allocation problems, the correctness rates drop to 43% and 38%, respectively.

Figure 6.27c illustrates the difference between the correctness rates of the XCS and the LUT in percentage points for easier comparison. The correctness rate of the XCS is always better than the correctness rate of the LUT. For the $(10, 3)$ and $(10, 5)$ allocation problems, the correctness rates of the XCS are 45 and 32 percentage points larger than the correctness rates of the LUT, respectively.

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

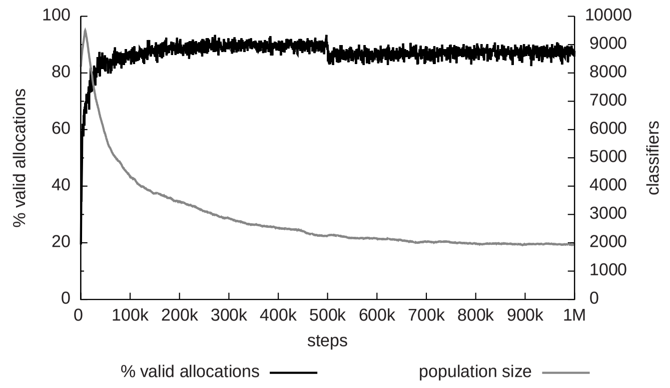
6.3.3 Generalization after restricted learning

In order to find the optimal action for every possible situation, the designer has to reason about it and/or simulate it. For the operator-allocation problem, the simulations run rather quickly, but simulations that take longer are easily imaginable, 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 time for design-time simulations at a minimum.

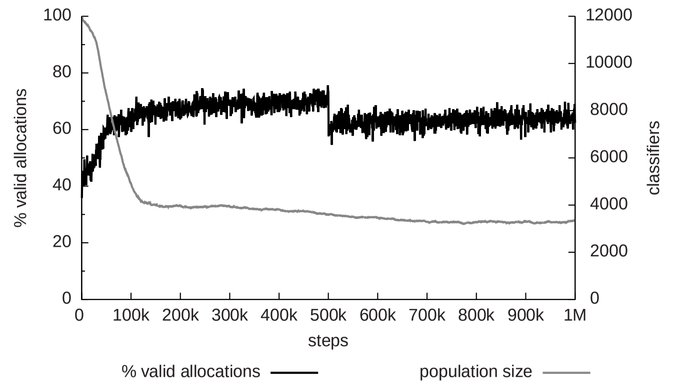
The last set of experiments covers both the design time and the run time of the self-adaptation system. At design time, the evaluator is presented only a randomly chosen, but fixed subset of all possible operator occupations. At run time, of course all operator-allocations are possible. Whether the XCS can generalize from the restricted design-time set to the complete run-time set is analyzed.

The results are shown in Figure 6.28. The first 500 000 learning steps represent the design-time learning. The following learning steps represent the run-time situation, where all operator occupations occur. In the top row, the XCS sees only a randomly chosen but fixed subset of 25% of all possible operator occupations. Figures 6.28a and 6.28b show the results for the $(10, 3)$ and $(10, 5)$ operator-allocation problem, respectively. The figure shows that during design-time learning, the correctness rates reach the usual 90% and 70% levels. After switching to run-time learning, the correctness rate drops slightly to about 85% and 65%, respectively, and stays there. The figure also shows that the variation of the correctness rate (the 'width' of the correctness rate line) is larger than when learning is based on all possible occupations. All other combinations of available and to be allocated operators are simulated, with similar results (not depicted). All combinations have a performance of what has been shown for the $(10, 5)$ allocation problem or better.

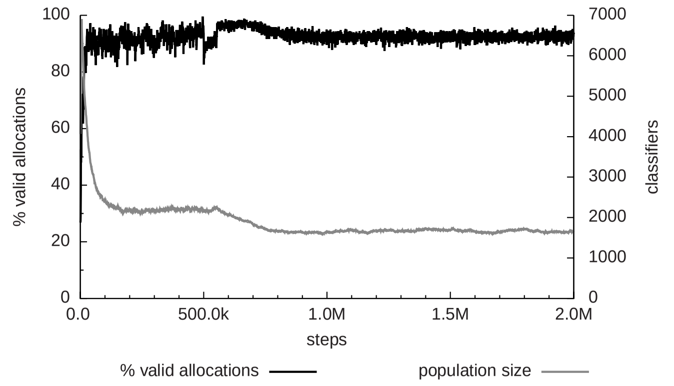
Learning is restricted further to find a lower bound for the amount of occupations needed during design-time learning. Figures 6.28c and 6.28d show the result for the $(10, 3)$ and $(10, 5)$ allocation problem, respectively. This time, the XCS sees only a random but fixed sample of 5% of all possible occupations. The figures show that the variation in the correctness rate increases considerably. For the $(10, 3)$ allocation problem, the variation



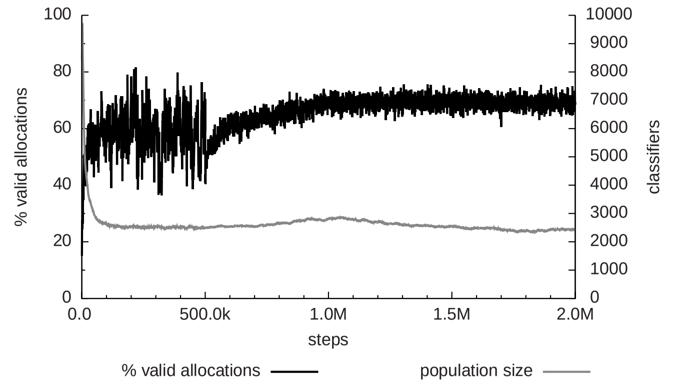
(a) $(10,3)$ allocation problem.



(b) $(10,5)$ allocation problem.



(c) $(10,3)$ allocation problem.



(d) $(10,5)$ allocation problem.

Figure 6.28: Run-time generalization from (top) 25%-restricted design-time learning and (bottom) 5%-restricted design-time learning.

Table 6.2: Optimal classifiers for the (4, 3) operator-allocation problem.

condition C	action a	reward p
000#	1	1000
1###	1	0
#1##	1	0
##1#	1	0
00#0	2	1000
1###	2	0
#1##	2	0
###1	2	0
0#00	3	1000
1###	3	0
##1#	3	0
###1	3	0
#000	4	1000
#1##	4	0
##1#	4	0
###1	4	0
11##	0	1000
1#1#	0	1000
1##1	0	1000
#11#	0	1000
#1#1	0	1000
##11	0	1000
000#	0	0
00#0	0	0
0#00	0	0
#000	0	0

of the correctness rate covers about 10%, while for the $(10, 5)$ 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 $(10, 3)$ allocation problem and 70% for the $(10, 5)$ allocation problem.

In conclusion, 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.

6.4 Validating the extended XCS theory

This section presents the results on validating the extended XCS theory with the example of solving the operator-allocation problem. Some of the results of this and the following subsection are published in the co-authored paper (Rakitsch et al., 2010).

6.4.1 Optimal classifiers

As already mentioned, the minimum set of classifiers that correctly predict the reward for every possible state and action is called the set of *optimal classifiers.* For example, Table 6.2 shows the optimal classifiers for the $(4, 3)$ operator-allocation problem. The following deduces the number of optimal classifiers for any (c, t) operator-allocation problem.

According to the formulation of the operator-allocation problem, its classifiers receive a reward of 1000, if the proposed action only allocates free operators, and a reward of zero, if one of the operators that is to be allocated is already occupied. First, consider the optimal classifiers that propose an actual allocation, that is, all classifiers proposing an action not equal to zero. Concerning the i bits in the condition that correspond to the operators that action a allocates, there is one classifier that has all bits set to 0 and thus receives the full reward and i classifiers that have one of the bits set to 1 and thus do not receive a reward. The other bits of the classifiers are set to the don't-care symbol #. Hence, there are $i+1$ optimal classifiers for each action $a \neq 0$.

Now consider the classifiers that propose that no allocation is possible, that is, classifiers with action $a = 0$. There is no possible action if $c - t + 1$ operators are occupied and there is a possible action if at least t operators are free. To the first set correspond $\binom{c}{c-t+1}$ classifiers that have $c - t + 1$ bits of their condition set to 1 and thus correctly propose action $a = 0$ and receive the full reward; to the second set correspond $\binom{c}{t}$ classifiers that have t bits set to 0 and thus wrongly propose action $a = 0$ and receive no reward. Hence, there are $\binom{c}{c-t+1} + \binom{c}{t}$ optimal classifiers that propose action $a = 0$.

Classifiers that only cover edge cases do not significantly contribute to the overall performance and thus are not strictly necessary. For the operator-allocation problem, this is true for classifiers that propose an allocation ($a \neq 0$) when proposing no allocation ($a = 0$) is correct most of the time. The probability that a classifier proposing action $a = 0$ receives a reward of 1000 is:

$$\Pr(\text{Action 0 gets reward 1000}) = \sum_{j=c-t+1}^{c} \binom{c}{j} 2^{-c} \tag{6.1}$$

Table 6.3 displays the probabilities for the individual operator-allocation problems with $1 \leq t \leq c \leq 10$. As expected, if many operators have to be allocated (t is large), choosing action $a = 0$ has a high probability of being valid. The bold numbers indicate when action zero gains a reward of 1000 in at least 90% of all cases. In these edge cases, optimal classifiers that propose an action $a \neq 0$ are not strictly necessary because always choosing action $a = 0$ is already enough to reach the required correctness rate of 90%. Therefore, these cases require only one classifier for each possible action with all condition bits set to the don't-care symbol. As the formulation of the equivalent problem dimension aims to ensure optimal classifiers, the problem dimension estimated in the following will be larger than necessary to achieve the desired correctness rate, because the formulation also covers the aforementioned edge cases. However, as the following will reveal, the corresponding k values are small, so the estimation is not far away from the optimum.

6.4.2 Estimating the equivalent problem dimension k

According to Equation (4.3), k_{avg} is calculated as the weighted sum of the individual classifier weights w_{cl}. The individual classifier weights are defined in Equation (4.9),

Table 6.3: Pr(Action 0 gets reward 1000) for the (c, t) operator-allocation problem.

$t \backslash c$	1	2	3	4	5	6	7	8	9	10
1	0.5	0.25	0.12	0.06	0.03	0.02	0.01	0.00	0.00	0.00
2		0.75	0.50	0.31	0.19	0.11	0.06	0.04	0.02	0.01
3			0.88	0.69	0.50	0.34	0.23	0.14	0.09	0.05
4				**0.94**	0.81	0.66	0.50	0.36	0.25	0.17
5					**0.97**	0.89	0.77	0.64	0.50	0.38
6						**0.98**	**0.94**	0.86	0.75	0.62
7							**0.99**	**0.96**	**0.91**	0.83
8								**1.00**	**0.98**	**0.95**
9									**1.00**	**0.99**
10										**1.00**

which is repeated here for convenience:

$$w_{cl} = k_{cl} \cdot \Pr(s) \cdot \left(P_{\text{explr}} \cdot \sum_s p_{\text{explore}}(s, cl) + P_{\text{explt}} \cdot \sum_s p_{\text{exploit}}(s, cl) \right) \tag{4.9}$$

In Equation 4.9, k_{cl} is the number of specified bits of the classifier cl, which can be determined by inspecting the classifier. As the input states are uniformly distributed in the operator-allocation problem, $\Pr(s) = (1/2)^c$. P_{explr} and $P_{\text{explt}} = 1 - P_{\text{explr}}$ are given as parameters to the XCS and are thus already known. To calculate w_{cl}, only $p_{\text{explore}}(s, cl)$ and $p_{\text{exploit}}(s, cl)$ remain to be determined. p_{explore} and p_{exploit} are calculated according to Equations 4.7 and 4.8, respectively, which are repeated here for convenience:

$$p_{\text{explore}}(s, cl) = \Pr(cl \in [A] \mid s, \text{explore}) \cdot \frac{1}{\|[A]_s^{a_o}\|} \tag{4.7}$$

$$p_{\text{exploit}}(s, cl) = \Pr(cl \in [A] \mid s, \text{exploit}) \cdot \frac{1}{\|[A]_s^{a_i}\|} \tag{4.8}$$

The following exemplifies the calculation of p_{explore} and p_{exploit} with the $(4, 3)$ operator-allocation problem. Table 6.4 shows some values of p_{explore} and p_{exploit} for $P_{\text{explr}} = 0.2$. For example, in input state $s_0 = 0000$, there are eight optimal classifiers that match. In explore mode, any of the matching classifiers may be selected into the action set. As the number of possible actions $n = 5$ for the $(4, 3)$ operator-allocation problem, $\Pr(cl \in [A] \mid s_0, \text{explore}) = \frac{1}{5}$ for any matching classifier cl according to Equation 4.10. For the four matching classifiers with $cl.a = 0$, $\|[A]_{s_0}^{a_o}\| = 4$, and thus $p_{\text{explore}} = \frac{1}{5} \cdot \frac{1}{4} = 0.05$. For the other matching classifiers with $cl.a \neq 0$, $\|[A]_{s_0}^{a_o}\| = 1$, thus $p_{\text{explore}} = 1 \cdot \frac{1}{5} = 0.2$. In exploit mode, there are four classifiers that predict the maximum reward, thus $n_a(s_0) = 4$ and $\Pr(cl \in [A] \mid s_0, \text{exploit}) = \frac{1}{4}$, according to Equation 4.11. As there is only one classifier for each action, $|[A]|_{s_0}^{a_i} = 1$ and $p_{\text{exploit}} = 1 \cdot \frac{1}{4} = 0.25$ for every classifier predicting the maximum reward. The values of p_{explore} and p_{exploit} at the other input states are calculated similarly.

Table 6.4: Values of p_{explore} and p_{exploit} for the $(4, 3)$ operator-allocation problem.

	cl in $[M]$				
s	$cl.C$	$cl.a$	$cl.p$	p_{explore}	p_{exploit}
0000	000#	0	0	0.05	0.00
	00#0	0	0	0.05	0.00
	0#00	0	0	0.05	0.00
	#000	0	0	0.05	0.00
	000#	1	1000	0.20	0.25
	00#0	2	1000	0.20	0.25
	0#00	3	1000	0.20	0.25
	#000	4	1000	0.20	0.25
0001	000#	0	0	0.20	0.00
	000#	1	1000	0.20	1.00
	###1	2	0	0.20	0.00
	###1	3	0	0.20	0.00
	###1	4	0	0.20	0.00
			⋮		
0011	##11	0	1000	0.20	1.00
	##1#	1	0	0.20	0.00
	###1	2	0	0.20	0.00
	##1#	3	0	0.10	0.00
	###1	3	0	0.10	0.00
	##1#	4	0	0.10	0.00
	###1	4	0	0.10	0.00
			⋮		
0111	#11#	0	1000	0.07	0.33
	#1#1	0	1000	0.07	0.33
	##11	0	1000	0.07	0.33
	#1##	1	0	0.10	0.00
	##1#	1	0	0.10	0.00
	#1##	2	0	0.10	0.00
	###1	2	0	0.10	0.00
	##1#	3	0	0.10	0.00
	###1	3	0	0.10	0.00
	#1##	4	0	0.07	0.00
	##1#	4	0	0.07	0.00
	###1	4	0	0.07	0.00
			⋮		
1111	11##	0	1000	0.03	0.17
	1#1#	0	1000	0.03	0.17
	1##1	0	1000	0.03	0.17
	#11#	0	1000	0.03	0.17
	#1#1	0	1000	0.03	0.17
	##11	0	1000	0.03	0.17
	1###	1	0	0.07	0.00
	#1##	1	0	0.07	0.00
	##1#	1	0	0.07	0.00
	1###	2	0	0.07	0.00
	#1##	2	0	0.07	0.00
	###1	2	0	0.07	0.00
	1###	3	0	0.07	0.00
	##1#	3	0	0.07	0.00
	###1	3	0	0.07	0.00
	#1##	4	0	0.07	0.00
	##1#	4	0	0.07	0.00
	###1	4	0	0.07	0.00

Table 6.5: Estimated k for the (c,t) operator-allocation problem.

$t\backslash c$	1	2	3	4	5	6	7	8	9	10
1	1.0	1.4	1.5	1.5	1.5	1.4	1.3	1.2	1.1	1.1
2		1.4	1.9	2.3	2.5	2.6	2.5	2.4	2.3	2.2
3			1.6	2.3	2.9	3.3	3.5	3.6	3.6	3.5
4				1.6	2.5	3.3	3.8	4.2	4.5	4.6
5					1.6	2.6	3.5	4.2	4.8	5.2
6						1.4	2.6	3.6	4.5	5.2
7							1.3	2.4	3.6	4.6
8								1.2	2.3	3.5
9									1.1	2.2
10										1.1

Table 6.6: Estimated N for the (c,t) operator-allocation problem.

$t\backslash c$	1	2	3	4	5	6	7	8	9	10
1	11	23	37	48	68	91	121	159	208	274
2		16	54	132	250	384	616	945	1 391	2 003
3			19	97	354	962	2 030	3 493	5 942	9 927
4				20	141	741	2 800	7 969	17 888	33 037
5					24	174	1 258	6 426	24 176	70 198
6						27	228	1 800	12 039	58 653
7							31	297	2 600	19 036
8								36	378	3 782
9									43	480
10										51

Given p_{explore} and p_{exploit}, the individual classifier weights w_{cl} and $k = k_{\text{avg}}$ can be calculated. Table 6.5 displays the resulting equivalent problem dimensions k for all (c,t) operator-allocation problems with $1 \leq t \leq c \leq 10$. For a given c, the equivalent problem dimension k first increases with t until about $c/2$, after which k decreases again. In fact, the problem dimensions for the (c,t) and $(c, c-t+1)$ operator-allocation problems are approximately the same, as these two problems are almost symmetrical concerning the conditions of their classifiers.

Comparing the estimations k_{avg} and k_{max}, k_{avg} is about half as large as k_{max}, resulting in considerably smaller population sizes.

6.4.3 Estimating the population size N

With the estimated problem dimension $k = k_{\text{avg}}$, the maximum population size N can be calculated. Analogous to the existing XCS theory, N shall be as small as possible, but large enough to fulfill the covering challenge, the schema challenge, and the reproductive opportunity, as described in Section 2.1.3. Table 6.6 shows the resulting population sizes.

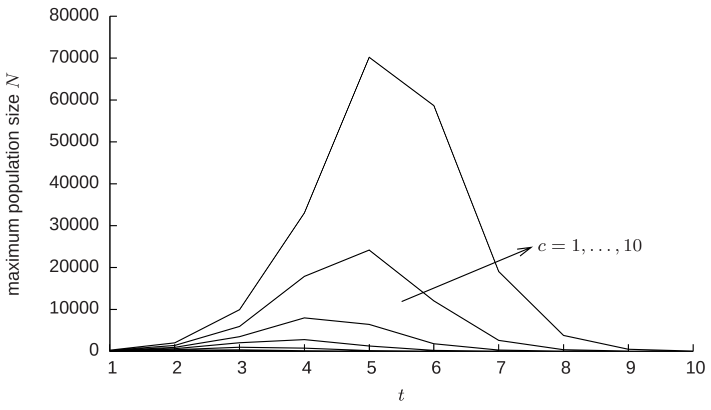


Figure 6.29: Derived maximum population size N of the (c, t) operator-allocation problem for $1 \leq c \leq 10$. The x axis shows the number t of operators to be allocated.

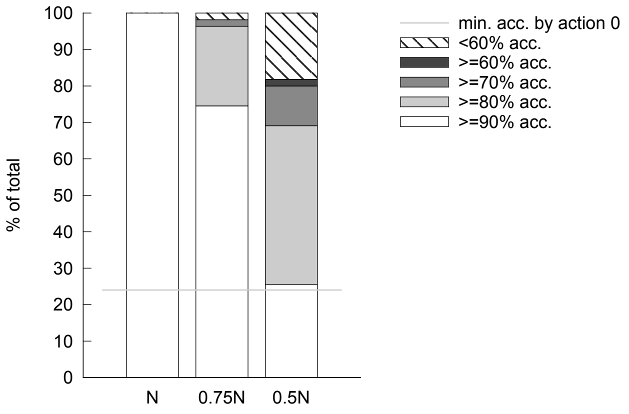


Figure 6.30: Percentage of solved operator-allocation problems at various accuracy levels, depending on N.

The results from Table 6.6 are plotted in Figure 6.29 for each c. For each number of operators c, the resulting maximum population size N first increases with the number of operators t that are to be allocated until about $c/2$, after which N decreases again. The figure shows that the maximum population size grows faster the larger c is. It also shows that N grows exponentially with c.

Figure 6.30 shows, for different maximum population sizes N, the percentage of all (c, t) operator-allocation problems with $1 \leq t \leq c \leq 10$ that the XCS solved at the displayed correctness level. The y axis depicts the percentage of the operator-allocation problems that where solved at the displayed accuracy level; the x axis shows the three different experiment sets with the population sizes N, $0.75N$ and $0.5N$. The figure shows that, wit the calculated population size N, the XCS can solve all operator-allocation problems with a correctness rate of at lest 90%. Reducing the maximum population size to $0.75N$ results in the XCS solving about 30% of the operator-allocation problems

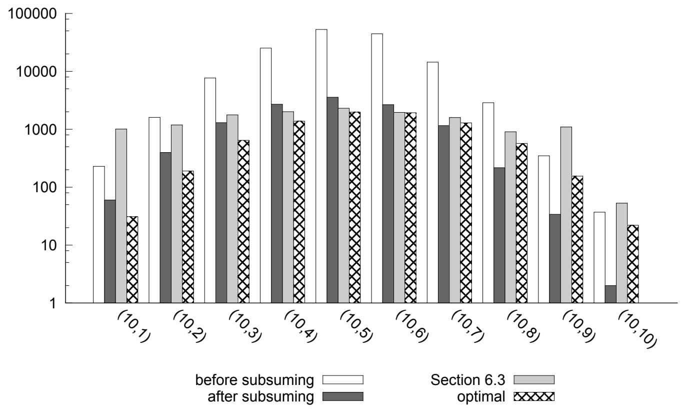


Figure 6.31: Number of classifiers for the $(10, t)$ operator-allocation problem.

with a correctness rate between 80% and 90%. Reducing the maximum population size further to $0.5N$ results in the XCS solving only 24% of the operator-allocation problems with a correctness rate of at least 90%, while in 20% of the experiments, the XCS has severe difficulties in choosing the right action and the correctness level drops below 70%. Repeated simulations give similar results.

Table 6.3 shows the probability that choosing action $a = 0$, which indicates that no allocation is possible, is the correct action of a random operator-allocation problem instance. In 13 out of 55 operator-allocation problems (i.e., in about 24% of the cases) always choosing action $a = 0$ results in a correctness rate greater than 90%. Since these problems can be solved without the optimal classifiers at a sufficiently high correctness rate, they require only a small classifier population. The black line in Figure 6.30 indicates the expected percentage of operator-allocation problems that can be solved even with very small maximum population size by just constantly choosing action $a = 0$. Excluding these "easy" problems, the figure shows that no problem from the experiment set with maximum population size $0.5N$ reaches a correctness level of over 90%.

Concluding, this section shows that the XCS is able to solve all operator-allocation problems sufficiently well only if it uses at least the maximum population size N that is derived from the extended XCS theory. Using smaller maximum population sizes leads to less or no problems solved. Thus, the derived maximum population size is an upper bound. From this follows, that the estimated problem dimensions k are also upper bounds, as they are large enough to solve operator-allocation problems, but not too large, since performance declines when N is reduced.

6.5 Subsuming after learning

Figure 6.31 displays the number of classifiers for the operator-allocation problems with ten operators. The x axis refers to the problem instances. The y axis shows to the number of classifiers on a logarithmic scale. The white bar depicts the number of classifiers before subsumption, the gray bar after subsumption, the black bar shows the number of classifiers from Section 6.3 and the patterned bar refers to the number of optimal classifiers. In Section 6.3, the XCS is applied to the same problem instances with the two methods *GA subsumption* and *Action Set Subsumption* activated; they both subsume during learning.

The figure shows that the subsumption method reduces the number of classifiers significantly. For the problems $(10, i)$ with $1 \leq i \leq 6$, the number of classifiers after subsumption is a bit larger than the number of optimal classifiers. For the problems $(10, i)$, $7 \leq i \leq 10$, the number of subsumed classifiers is smaller than the number of optimal classifiers. This can be attributed to action $a = 0$, as in these cases selecting action zero most of the time regardless of the system input suffices to achieve the desired performance level of 90% or more (see Table 6.3). As t approaches c, the probability that choosing action $a = 0$ results in the maximum reward increases. Because in these cases, not all optimal classifiers are required, the number of classifiers can drop below the number of optimal classifiers.

The figure also shows that for small and large t, subsuming during learning results in more classifiers than subsuming after learning but still less classifiers than without subsumption. For medium-sized t, subsuming during learning and after learning results in about the same number of classifiers.

For problems with $c < 10$, the results are similar. In the worst case, the $(2, 1)$ operator-allocation problem, only 17% of the classifiers are subsumed. In the best case, the $(10, 10)$ operator-allocation problem, 95% are subsumed, as previously mentioned, mainly due to the effect of action $a = 0$. On average, the number of classifiers can be reduced to 27% of classifiers before subsumption.

Summarizing, subsumption after learning reduces the number of classifiers to a magnitude that is comparable to the magnitude of the number of optimal classifiers. With GA subsumption and action set subsumption, at least as many classifiers are needed as when subsuming after learning. For many problem instances, subsumption after learning reduces the number of classifiers more than subsumption during learning.

6.5.1 Solving the operator-allocation problem after subsumption

The next experiment set, uses the classifiers from Figure 6.31 as initial classifiers.

Figure 6.32 shows the correctness rates without subsumption, with subsumption after learning and with subsumption during learning with the methods GA subsumption and action set subsumption, as done in Section 6.3. The figure repeats the result from Section 6.3, that all operator-allocation problems can be solved with the non-subsumed classifiers. When subsuming after learning, in 20% of the problems the correctness rate drops to a level between 80% and 90%. When subsuming during learning, in 20% of the problems the correctness rate drops to a level between 80% and 90% and in 20% even to a level below 80%.

In conclusion, subsuming after learning requires no more classifiers than subsuming during learning, but the correctness rate remains on a higher level. However, subsuming after learning comes with a small loss in correctness rate, traded for a smaller number of classifiers.

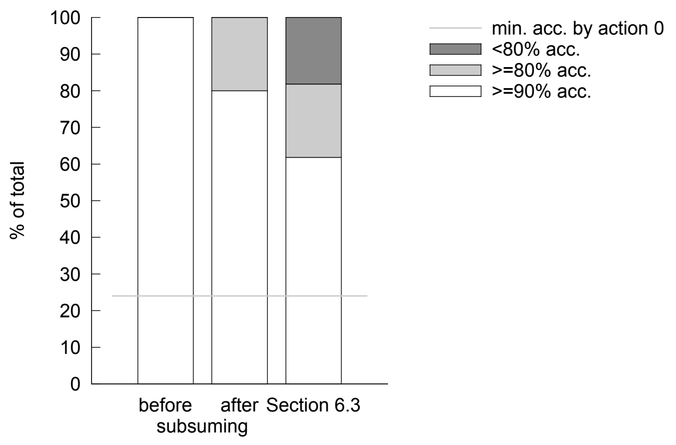


Figure 6.32: Correctness with subsumed classifiers.

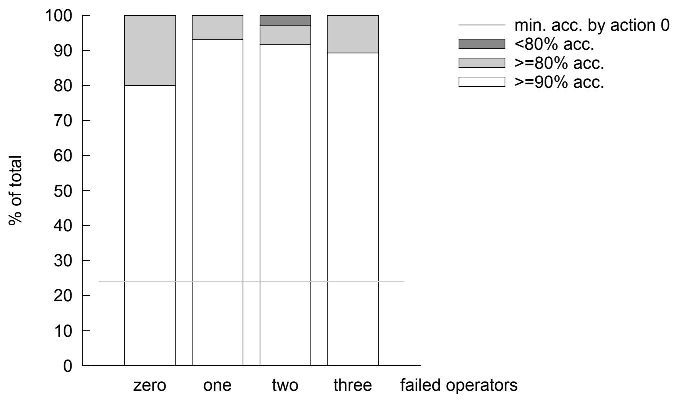


Figure 6.33: Correctness rate of the XCS with subsumed classifiers when operators fail.

6.5.2 Self-adaptation to failing operators

Section 6.3 shows that the XCS is able to self-adapt to the unforeseen event of operator failure when GA subsumption and action set subsumption are activated. This section investigates if the XCS maintains this capacity when the newly introduced subsumption method is used instead. For this, one or more operators are set to fail and the XCS has to adapt to the new situation. As mentioned previously, a failed operator is still monitored as free, but cannot be allocated.

The XCS runs the first 150000 time steps normally before the first operator fails. After another 100000 time steps, the next operator fails and so forth. In the (c, t) operator-allocation problem, at most $c - t$ operators can fail, since after that, no valid allocation is possible anymore.

Figure 6.33 shows the correctness rate of the XCS with subsumed classifiers when no operator, one, two, or three operators have failed. The figure shows that the correctness rate nearly stays the same after one to three operators have failed. The average correctness

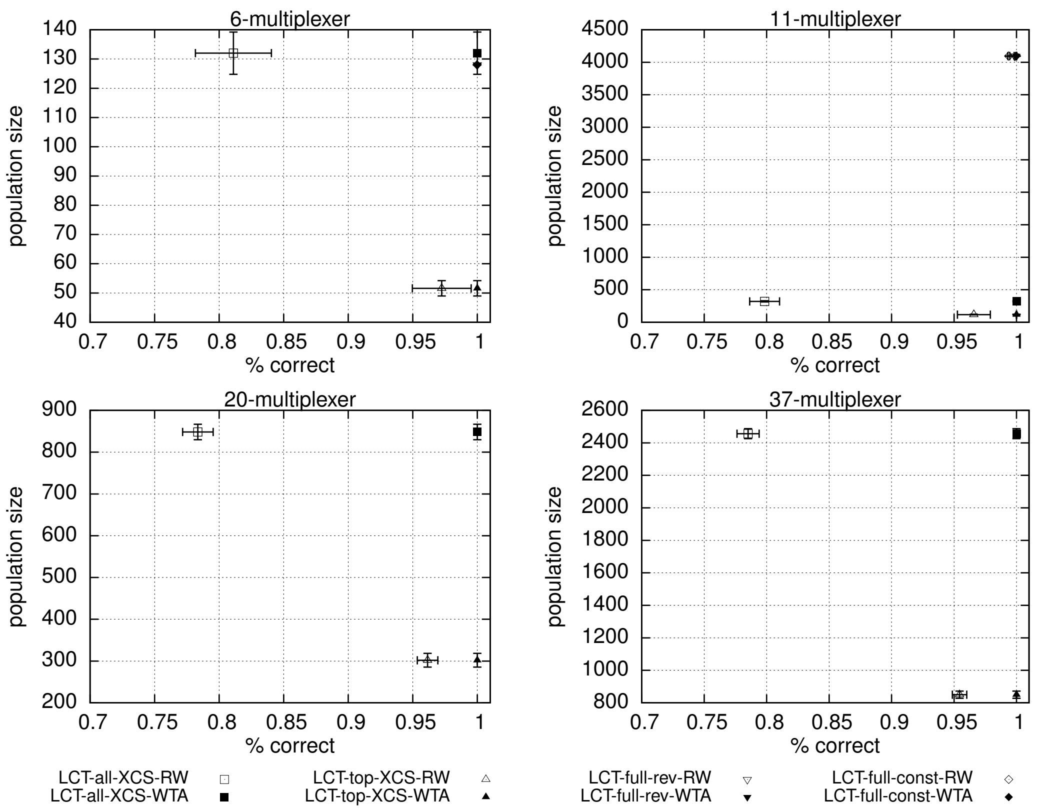


Figure 6.34: Performance in the multiplexer problems. *Clockwise from upper left:* 6-, 11-, 20-, and 37-multiplexer. Within each graph, lower right is better. Note that the y-axes differ in scale.

rate is between 94% and 97% in all experiment sets. Hence, the new subsumption method does not affect the self-adaptation capacities of the XCS.

6.6 Running in hardware

This sections presents the results on the two problem types multiplexer and operator-allocation. The results are published in Bernauer et al. (2010).

6.6.1 Solving the multiplexer problem

Figure 6.34 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. Here, the top-XCS translation uses only classifiers that predict the maximum reward with perfect accuracy. As a small but correct LCS is desirable, in each graph lower right is better. The figures show that in the new

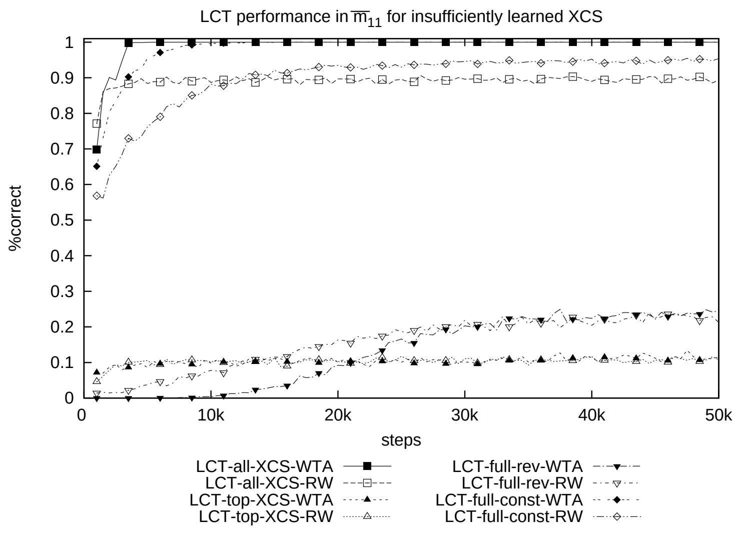


Figure 6.35: LCT performance in the inversed multiplexer problem $\overline{m}_{11}$ using rules from an insufficiently learned XCS.

winner-takes-all (WTA) strategy (solid 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 by Bolchini et al. (2006). 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, they are excluded from the 20- and 37-multiplexer problem.

All LCT configurations are able to perfectly adapt to the unforeseen 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 is able to solve the multiplexer problem sufficiently well (e.g., because XCS' learning process has not been terminated prematurely). Otherwise, even if the XCS shows a correctness rate of 100%, not all LCT configurations can adapt to the inversed multiplexer. Figure 6.35 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. Presumably, 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, the LCT shows that with the all-XCS translation, it is possible to achieve a high correctness rate and to retain the capability to adapt to unforeseen events. The full-const translation shows similar results. Combining

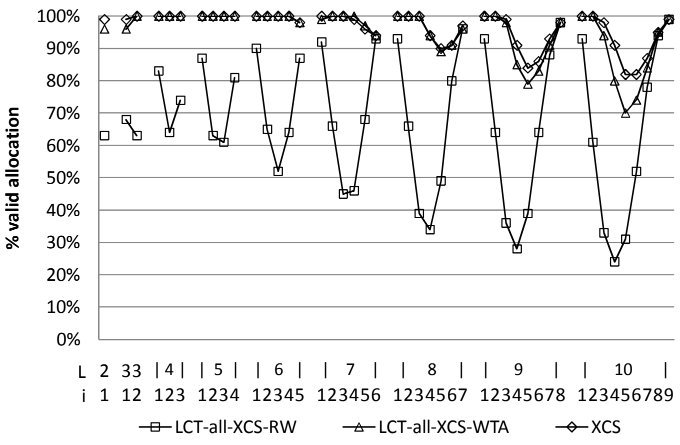


Figure 6.36: Rate R_{LCT} of valid operator-allocations in the LCT and R_{XCS} for comparison.

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 for the top performing XCS rules from the top-XCS translation. As for more real-world problem types the XCS cannot always learn perfectly, the following experiments concentrate on the all-XCS translation.

6.6.2 Solving the operator-allocation problem

Figure 6.36 shows the rate R_{LCT} of valid operator-allocations of the LCT for the (c, t) operator-allocation problems, $1 \leq t < c \leq 10$, 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}, the latter for comparison. The figure shows that the LCT uses rule bases for which the XCS correctly allocates more than 90% of the problem instances for $c < 9$ and more than 80% for $9 \leq c \leq 10$, comparable to what has been reported in Section 6.3. The LCT using the winner-takes-all strategy (WTA) has very similar rates to the XCS, with a larger difference only for $c = 10$. Using the roulette-wheel strategy (RW), the LCT finds valid allocations considerably less often; in particular for $1 < t < c - 1$, R_{LCT} drops as low as 22%. The reduced performance in the $(10, 5)$ and $(10, 6)$ problem instances concurs with the findings in Section 6.3 that these two problem instances are the most difficult for the XCS.

To test LCT's ability to adapt to unforeseen events, the LCT is initialized with the all-XCS-translated XCS rules and the operators are set to 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; the XCS is depicted solely for comparison purposes.

Figure 6.37 shows R_{LCT} and R_{XCS} after the first (left half) and the second (right half) randomly chosen operators have failed. Note that the diagram shows fewer problem instances for the second operator failure, as not every instance allows the failure of two operators (e.g., when allocating three operations out of four operators, the failure of two operators turns the problem unsolvable). The rate of valid operator-allocations of the

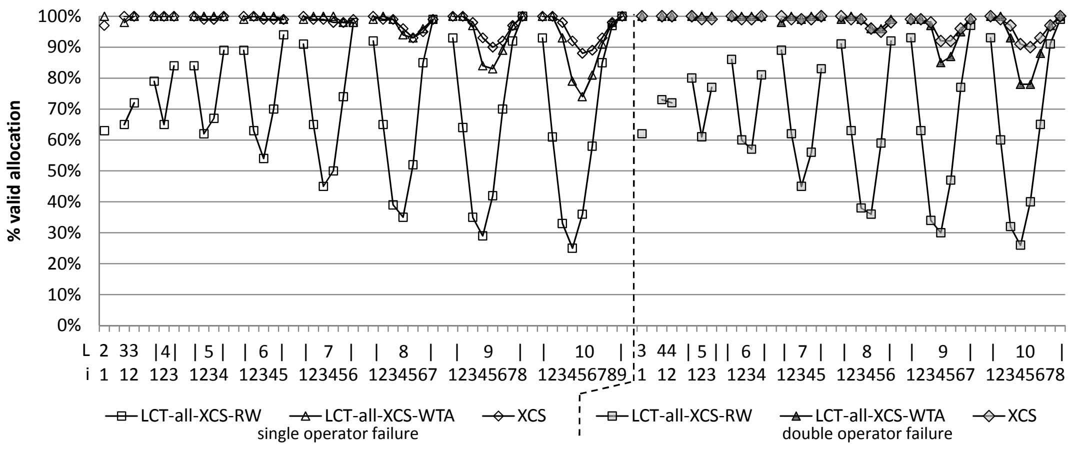


Figure 6.37: Rate R_{LCT} of valid operator-allocations of the LCT after one or two randomly chosen operators have failed, and R_{XCS} for comparison.

LCT increases slightly, on average by about 1 percentage point (maximum 10 percentage points) after the first operator has failed and an additional 1 percentage point (maximum 11 percentage points) after the second operator has failed. Compared to the rates before any operator has failed, an increase of about 2 percentage points on average (maximum 17 percentage points) can be observed. The increase is of about the same amount for any employed action selection strategy, with the roulette-wheel strategy showing a greater variance. The results show approximately the same increase that the XCS would show. As reported in Section 6.3, the valid operator-allocation rate generally increases after an operator fails because the probability that the action "no valid allocation possible" is correct increases.

Summarizing, with the newly introduced winner-takes-all action selection strategy, the LCT shows rates of valid operator-allocations which are comparable to what can be found with the XCS. The LCT also retains the capability to adapt to the unforeseen failure of up to two operators. The roulette-wheel strategy, however, shows high rates of valid operator-allocations only for some border cases.

6.7 Costs

A self-adaptation methodology involves costs during design time and additional chip area, of course. Quantifying the costs involved during design time is difficult, though. In any case, it can be qualitatively argued why 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

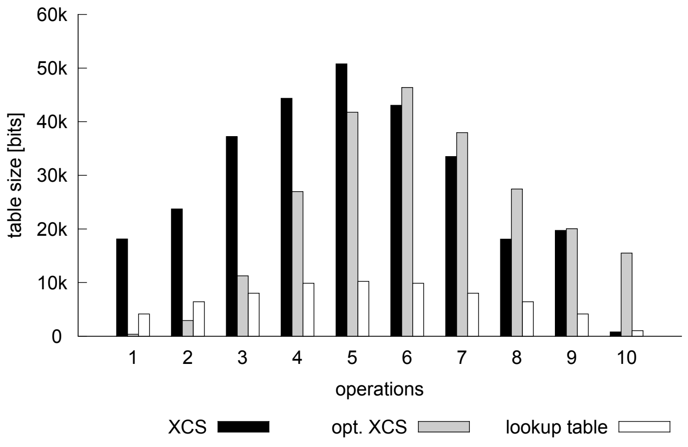


Figure 6.38: Table sizes for the XCS, the optimal XCS and the lookup table.

output vectors, and the design has to be changed accordingly. This is a straightforward task in the LCT, as it involves only regular structures (see also Section 2.1.5). 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, 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 no experiments have been conducted for this yet, the resulting classifier tables of the experiments that are presented here and by Kovacs (1997), who reports that the XCS evolves complete and minimal rule representations, gives a reason for this expectation. For the actual table sizes, see the following.

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 that are necessary to get the assessment of the achieved state of the chip. Although the complexity of this task 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, it is expected that the design of a calculating circuit is less difficult than the design of a special-purpose circuit to solve an adaptation problem.

The necessary chip area to implement the XCS is mainly determined by the number of classifiers that the XCS has generated to obtain the correctness rates reported previously. The size of a preliminary implementation of the LCT, a possible hardware version for the self-adaptive controller, is given in Section 2.1.5; what remains to be estimated is the size of the memory that is required to store the classifiers. From the experiments on the basic operator-allocation problem (see Section 6.3.1), the population size is collected after the correctness rate did not improve any further. Figure 6.38 shows the resulting population sizes for the problems with a total of $c = 10$ operators. The x axis identifies the number t of operators 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 (black), a theoretically optimal XCS (labeled "opt. XCS", gray), and a fixed lookup table (white). For the actual XCS, each classifier consists of $c = 10$ bits for the condition, $\left\lceil \log_2 \left(\binom{c}{t} + 1 \right) \right\rceil$ bits for the action[2] and four bits for the predicted reward. The optimal XCS contains the minimum amount of classifiers that guarantees a correctness rate of 100%, representing the lower bound for the size of a classifier table for 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[3].

Figure 6.38 shows that the XCS needs about five times as many bits than a lookup table. For small numbers of t, 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 show 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).

[2] One action more than possible combinations is needed for the special action when there is no valid allocation.

[3] Or, more precisely, it contains the index of the optimal action.

7 Conclusions and future work

This thesis proposed a design methodology for a self-adaptive controller to realize self-adaptation at chip level. The increasing design complexity of SoCs requires high design reuse rates to keep the time to market short. Current adaptive circuits help to mitigate process variation and other variabilities, but their design cannot be easily reused, they take a considerable amount of time during the design process, and they are closely intertwined with their sensors and effectors. The proposed self-adaptive controller overcomes these shortcomings with a machine learning algorithm and two different versions at design time and at run time. Three application examples have shown that the proposed self-adaptive controller is reusable in different designs (be it a control problem or a combinatorial problem), that the proposed design methodology automates the design of the controller, and that the controller operates with different sensors and effectors. Additionally, the self-adaptive controller adapts to unexpected events, scales with multiple cores without the need for central control, and generalizes over restricted training sets. The feasibility of a realization in hardware has been presented. The current theory of XCS has been extended to cover problems that have been encountered during the development of this thesis and a new action selection method for the LCT has been suggested.

The presented benefits are achieved mainly due to the machine learning algorithm: instead of manually adjusting the controller, the machine learning algorithm learns the beneficial actions for different chip conditions, both at design and at run time. Unlike other machine learning algorithms, the learning classifier system XCS requires only a manageable amount of hardware overhead and its acquired knowledge can be reused in different designs. The benefits have been shown in the application examples, which are discussed in the following.

7.1 Discussion

The first application example shows that the proposed self-adaptation system can actually control the operating conditions of a chip (here: frequency and voltage) and that the controller adapts itself to environmental conditions that differ from the expectations at design time (raised ambient temperature and exchanged reward function). However, under certain circumstances the system oscillates, a property that is regularly observed with controllers, albeit here due to different reasons. One reason is that the XCS "forgets" that the previously chosen setting makes the system temperature raise above limits. Yet, the behavior is to be expected from a system that constantly tries to optimize the current operating point: stopping to raise the frequency also means stopping to test if the environment has changed such that a higher frequency is possible. The amplitude of the oscillation is governed by the discretization of the inputs and outputs; smaller discretization steps will lead to smaller amplitudes. Incorporating an additional action–result memory to the XCS might avoid some oscillation, but only down to a certain frequency of occurrence.

The second application example shows the scalability of the self-adaptive controller with many, highly interacting components: each additional component requires only one additional self-adaptive controller—a controller placed at a higher level for coordination is not necessary. While isolated self-adaptive controllers fail to control the many cores of the simulated MPSoC, cooperating self-adaptive controllers achieve this goal by exchanging classifiers. This emergent property of the system is due to the increased selection pressure in the niches of the isolated populations. The distributed self-adaptation system still adapts to changing environmental conditions. Several different possible configurations of the cooperating, distributed self-adaptation system have been examined. Emigrating classifiers by fitness, deleting them by prediction error and a fully connected topology for communication turned out to be the best configuration in this application. However, this optimal configuration is only discovered by testing all possible configurations. A suitable search strategy could mitigate the effort to find optimal configurations for cooperation.

The results of the third application example of the operator-allocation problem show the broad applicability and flexibility of the proposed self-adaptive controller. The self-adaptation system is applicable not only to control problems, but also to combinatorial problems. The flexibility of the proposed system is shown by generalizing from a restricted learning set: given only a subset of all possible input states at design time, the self-adaptation system can generalize so that it covers all input states at run time; however, the generalization results might be due to the combinatorial nature of the operator-allocation problem and not directly transferable to other problems. The results are significantly better than random allocation. Once operators start to fail, the proposed system tolerates these unexpected events and maintains its high correctness rate. However, while the rate of valid allocations is usually above 90% (which might be enough for some applications if a short time to market can be achieved), it is never perfect. This is in contrast to lookup tables, which, by design, always allocate correctly. Yet, in case of operator failures, the performance of the lookup-table drops considerably, as it has no self-adaptation capabilities. But even a lookup table with self-adaptation capabilities such as Aron can hardly outperform the proposed system, as the direction of adjustment is unknown for combinatorial problems. Only a manually devised algorithm could be expected to perform better; however, this defeats the purpose of this thesis to reduce the manual effort of special-purpose solutions and to provide a reusable controller.

Additionally, this thesis introduces the equivalent problem dimension for complex problems to cover the new applications, extending the current state of XCS theory. Experimental results validate the extended theory: the calculated maximum population sizes are sufficiently large so that the XCS can solve the operator-allocation problems, but not too large (at least not by more than 25%), as smaller population sizes diminish the performance of the XCS. Like with current XCS theory, the presented extended XCS theory still needs knowledge of the optimal classifiers of the problem at hand. As the XCS has been shown to not be very sensitive in the selection of its parameters, the parameters of problems for which the optimal classifiers are not known can be derived from similar problems for which the optimal classifiers are known.

Subsuming after learning, that is, between design and run time, reduces the overall number of classifiers that have to be stored on the chip. The subsumption method reduces the number of classifiers considerably with only a small impact on the performance of the self-adaptation system. The classifiers retain their ability to adapt to the unexpected event of operator failure.

The LCT is a feasible hardware implementation for the proposed self-adaptive controller. The proposed design methodology can learn suitable classifiers for the LCT such that the performance at run time can be almost as good as the performance at design time or as what could have been achieved with a software solution. Furthermore, this thesis proposes to use the winner-takes-all action selection strategy in the LCT for improved results. The hardware realization retains the ability to adapt to the unexpected event of up to two operator failures.

7.2 Preconditions and limitations

The preconditions and limitations of the proposed design methodology for a self-adaptive controller are detailed in the following.

One precondition of the proposed self-adaptive controller is a suitable design-time model of the final design as it is available in current designs. As the self-adaptive controller can compensate some inaccuracies, the design-time model does not have to be overly accurate. Still, a design-time model is necessary for design-time learning. In this thesis, SystemC models have been used throughout. Other simulation models, such as Simulink, are possible if they can connect to the XCS.

Further, a suitable cost or reward function must be available. The reward function should not only measure the “goodness” of the chip, but should also be realizable with as little hardware as possible and, in the ideal case, guide the learning process of the XCS. The reward functions of the application examples only use operations that require little hardware, such as addition or subtraction. If the weights are chosen accordingly, costly operations such as multiplication or division can be approximated with shift operations or small lookup tables, which might reduce resolution, though. Designing a calculating circuit is standard practice and not expected to overly prolong the design process.

Last, appropriate sensors and effectors must be available, of course. As the self-adaptive controller has no model of its inputs and outputs, using any sensor that provides a digital signal is without problems. The same is true for effectors; however, effectors might be shared with other systems. For example, modern operating systems also adjust the frequency and voltage of the hardware components. This may interfere with the operation of the self-adaptive controller, either directly (e.g., both use the same component to adjust frequency or voltage), or indirectly (e.g., both try to control temperature through different means). As having the operating system in the control loop of the self-adaptive controller defeats the purpose of an autonomic and quick response of the hardware, an obvious solution is to put the self-adaptive component in command and let the operating systems signal its desired adaptations as an additional input signal. While this avoids that the additional knowledge of the operating system is not lost for self-adaptation, this is a major departure from standard practice, which asks for more research work.

Concerning the costs of the proposed design methodology, the qualitative analysis has shown that they seem manageable, as the methodology is built on reusable components. However, a quantitative analysis is missing. The largest costs are the design and hardware implementation of the reward function as well as the chip area that is necessary to store the classifiers.

As the self-adaptive controller continues to learn at run-time and learning involves making mistakes, the device must be able to cope with (spurious) mistakes. Possibly, mistakes are tolerable to achieve a shorter time to market as long as the device operates optimally otherwise (e.g., a video playback system that sometimes has faulty pixels not visible to the eye but that requires very little power). Alternatively, additional guarding circuit is added that avoids undesirable settings. While guarding circuits increase the design effort, this approach still avoids that the designer has to specify the optimal parameter settings; it can be expected that the overall effort is less than designing new adaptive circuits from scratch for each new adaptation problem.

A limitation of the proposed design methodology for the self-adaptive controller are that learning in the XCS is based on visiting all possible input states. If the input states are numerous, the time needed for design-time learning increases, prolonging the design process. However, as the third application example has shown, at least for some applications the XCS can successfully generalize from a restricted input-state set. Future work has to discover what kind of applications are suitable for generalization to reduce the time needed for design-time learning. The work of (Fredivianus et al., 2010) indicates there could be more than one would think initially.

A further limitation is that the XCS does not achieve a 100% correctness rate for some applications. This is partly due to the XCS trying new settings to increase the reward. Another cause is that the XCS employs a steady-state (overlapping) genetic algorithm, which is known to have a large variance (Bäck et al., 2002, Chapter 28). A common approach to mitigate this effect is to increase the population size, which unfortunately runs against the goal to use as few classifiers as possible to keep required chip area small. An alternative to larger population sizes is to use a generational (non-overlapping) genetic algorithm, which shows less variance. However, no application of the XCS with a generational genetic algorithm is known, so this has to be done in future work.

If the actions of the self-adaptive controller are meant to correct hard errors or soft errors (as in the second application example), the question arises how well the self-adaptive controller itself is protected against such errors. The robustness of the self-adaptive controller has not been investigated in particular in this thesis. The structure of the XCS, however, allows to assume a robustness against soft errors. As the values of the classifier parameters such as predicted reward or accuracy do not solely depend on one single execution of the classifier, the effect of intermittent soft errors can be expected to be marginal. If the soft-error rate is low, the learning during subsequent runs will correct any wrongly attributed rewards. The maximum allowable soft-error rate has to be determined in future work.

While the XCS can be expected to tolerate soft errors by its learning capabilities, hard errors may have more severe consequences. Assuming the LCT as the hardware implementation of the XCS, a hard error during matching and action selection may lead to a single classifier or a fixed set of classifiers being executed repeatedly, for example due to a stuck-at-zero error in the relevant register. In this case, the reward distribution will be flawed and cannot be corrected by subsequent learning. The result will be a malfunctioning self-adaptive controller. Fortunately, hard errors occur rarely during the run-time of chip and are mostly detected during testing after production. If hard errors are an issue, special hardening of the self-adaptive controller is advised. Hardening only the self-adaptive controller instead of the whole device may be a cost-effective way to reliable SoCs, a technique that can be investigated in future work.

Although this thesis has shown that the self-adaptive controller has many benefits over simple adaptive circuits, self-adaptive and adaptive circuits share the same limitations concerning testing after production. On the one hand, it is difficult to test the (self-) adaptive properties of the controller; on the other hand, it is difficult to test other chip components if an adaptive controller is present. Testing the controller itself is difficult as the controller expects to be executed in a running system while during testing the clock is regularly halted. With a stopped clock, the (self-) adaptive controller cannot react to changes in the environment, such as a raise or drop in temperature. Furthermore, the test environment may differ significantly from the production environment as testing executes operations that are atypical for production, such as a read-in of test vectors or voltage drops due to a stopped clock. Testing other components can be difficult, too, as the reaction of the controller can interfere with the test conditions. For example, while testing components with different voltage levels, the controller may readjust the voltage level. A possible solution is to turn off the controller during these tests; however, in this case the test environment differs from the production environment, which may negatively affect the validity of the test results. Testing of adaptive circuits is ongoing research work, into which self-adaptive controller should be included.

7.3 Future work

Besides addressing the limitations mentioned previously, such as interfacing the operating system or employing a generational XCS, the proposed design methodology can be improved by the following future work.

While learning and calibration of the self-adaptive controller happens automatically, simulations still take a considerable amount of time. For example, even for the simplified MPSoC simulation model, learning one configuration takes about one day on current hardware. A major bottleneck is the temperature simulation with HotSpot, which currently runs single-threaded. HotSpot executes the fourth-order Runge–Kutta method to solve the set of differential equations, which can be accelerated with machine-architecture specific basic linear subprograms (BLAS) such as Intel's Math Kernel Library or AMD's Core Math Library. Although BLAS reduces computation time in HotSpot by about 30%, according to own measurements, parallelizing over several cores may reduce computation time further. Similarly, for elaborate models, parallel SystemC simulations (Schumacher et al., 2010) may also reduce computation time. Speeding up computations has not been the focus of this thesis and thus left over as future work. Until then, the generalization capabilities of the self-adaptive controller may already help to reduce simulation time.

Another way to reduce simulation time, at least for cooperating self-adaption, is to reduce the number of necessary parameter configurations that have to be tested until an optimal configuration is found. In the present work, almost all possible combinations have been tested. Future work may identify the effect of the parameters on the performance of cooperating self-adaptation and develop a search strategy such that fewer parameter configurations have to be tested; essentially, a method to efficiently explore the design space of cooperating self-adaptation is desired.

The fact that the behavior of the self-adaptive controller is mainly defined by the classifiers raises the interesting possibility to manipulate the classifiers at run time by an external system. For example, a special set of classifiers can be loaded at run time for diagnostic or other testing purposes, raising special debugging signals when the monitors observe more or less complex chip conditions. Further, the manufacturer can provide a classifier update for an improved performance of the chip or as a bug fix, possibly based on feedback that has been recorded by the chip. Alternatively, different classifier sets can be shipped for different markets, different market segments (such as premium or discount), or different users, addressing the individual optimization criteria. Future work will explore the benefits and possibilities of run-time classifier updates, in particular what kind of feedback the chip should record such that the chip's performance can be increased by a classifier update.

Updating and improving the parameters of the design after the evaluation step is currently executed manually in the design process. Future work must show how the design methodology can automate this step or at least assist the designer in choosing new parameters for a new evaluation cycle. Other future work will involve how the memory footprint of the classifiers can be reduced further, besides subsuming after the design-time stage, for example with the algorithms of Wilson (2002) and Dixon et al. (2003).

A possible enhancement in the LCT is to alternate the action selection strategies. The current results use either the winner-takes-all or the roulette-wheel action selection strategy. Alternating the action selection strategies at run-time as it is done at design-time might be beneficial to run-time learning. Additionally, the new discovery component presented by Fredivianus et al. (2010) may be a suitable candidate to extend the LCT, which is currently missing such a component.

The present work has assumed that the hardware realization of the self-adaptive controller offers enough input and output ports. Designers can use off-the-shelf versions of the self-adaptive controller with a fixed number of input and output ports to avoid adjusting the self-adaptive controller repeatedly, which thwarts the attempt to reduce design effort. As the XCS is able to ignore input signals that do not influence the reward, probably only a few number of versions are necessary; for example, a self-adaptive controller with 8-bit input can be used with sensors providing only 5-bit input. These suppositions can be confirmed in future work, which would also have to quantify the associated power overhead, leading us closer to self-adaptation at chip level as part of a regular SoC design process.

Publications

This thesis is based on the following publications of the author:

- Bernauer A, Fritz D, and Rosenstiel W. Evaluation of the learning classifier system XCS for SoC run-time control. *Lecture Notes in Informatics (LNI)*, volume 134, pp. 761–768. Gesellschaft für Informatik, Springer. 2008.
- Bernauer A, Bringmann O, and Rosenstiel W. Generic self-adaptation to reduce design effort for system-on-chip. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 126–135. 2009.
- Rakitsch B, Bernauer A, Bringmann O, and Rosenstiel W. Pruning population size in XCS for complex problems. *World Congress on Computational Intelligence (WCCI)*, pp. 3383–3390. IEEE, Barcelona, Spain. 2010.
- Bernauer A, Zeppenfeld J, Bringmann O, Herkersdorf A, and Rosenstiel W. Combining software and hardware LCS for lightweight on-chip learning. *Distributed Parallel and Biologically Inspired Systems (DPBIS), IFIP AICT 329*, pp. 279–290. Springer. 2010.
- Sander B, Bernauer A, and Rosenstiel W. Design and run-time reliability at the electronic system level. *IPSJ Transactions on System LSI Design Methodology (TSLDM)*, volume 3, pp. 140–160. 2010.

List of Figures and Tables

List of Figures

List of Tables

Glossary of symbols, variables, and functions

Learning classifier systems

C	set of possible conditions
A	set of possible actions
P	set of possible rewards
cl	classifier
$cl.c$	classifier's condition
$cl.r$	classifier's expected reward (also called 'prediction')
$cl.f$	classifier's fitness
$cl.as$	classifier's average action set size
$cl.ts$	classifier's time stamp of last GA participation
$cl.exp$	classifier's experience
$cl.num$	classifier's numerosity
$cl.\varepsilon$	classifier's prediction error
l	condition length
$\#$	don't-care symbol
$\sigma(cl)$	specificity of cl, the ratio of the number of '0' and '1' to the number of '#' in the $cl.c$
k	number of '0' and '1' in classifier condition (number of specified bits, called 'schema order' or 'problem dimension')
s	state or problem instance
$[P]$	classifier population
$[M]$	match set
$[M]\|_a$	set of all classifiers in the match set that propose action a
$[A]$	action set
$P(a)$	prediction array
ρ	received reward
κ', κ	relative (numerosity-weighted) and current absolute accuracy
ε_0	classifiers with $cl.\varepsilon < \varepsilon_0$ are considered accurate
ν	parameter to calculate accuracy
θ_{sub}	classifiers with $cl.exp > \theta_{\text{sub}}$ are considered sufficiently experienced
P_{explt}	probability of exploit mode in LCS
P_{explr}	probability of explore mode in LCS
$\pi_{\text{LCS1}}(s)$	action selection strategy of LCS1, given state s (Equation (2.1), page 15)
β	learning rate

γ	discount factor
$P_{\#}$	don't-care probability
θ_{GA}	number of learning steps between GA invocations
p	classifier's parent
χ_{GA}	crossover probability
μ_{GA}	mutation probability
N	maximum population size
θ_{del}	classifiers with $cl.exp > \theta_{\mathrm{del}}$ are considered experienced enough to be a deletion candidate
δ	experienced classifiers with $cl.f < \delta\bar{f}$ are deleted, where $\bar{f}$ is the average fitness of the population
$\Pr(\text{cover})$	probability that at least one classifier covers a given input state
$\Pr(\text{match})$	probability that a random classifier matches a given input state
$s([P])$	average proportion of specified bits in the population

Physical models

P_{total}	total power dissipation
P_s	static power dissipation
P_d	dynamic power dissipation
V_{DD}	supply voltage
$k_{\mathrm{design}}, \hat{I}_{\mathrm{leak}}$	design- and technology-dependent parameters
α	activity; average number of zero-to-one transitions during a clock cycle
C_L	lump capacitance
f	clock frequency
$t_{\mathrm{r}}, t_{\mathrm{f}}$	signal rise and fall times at a transistor
t_{dT}	influence of temperature on time delay
β_{n}, β_{p}	gains of NMOS and PMOS
V_{tn}, V_{tp}	NMOS and PMOS threshold voltages
d	constant in Zhu's model of soft errors, usually less than five
λ_0	average fault rate at maximum voltage and frequency in Zhu's model of soft errors
C_{convec}	convection capacitance in temperature model
R_{convec}	convection resistance in temperature model
l, w_{sink}	length and width of heat sink
l, w_{spreader}	length and width of heat spreader
N_{inv}	number of inverters when simulating timing errors

Self-adaptive controller

n_i, n_o	number of sensors (monitors) and effectors (actuators) of the self-adaptive controller
$\mathcal{M}$	parameter of the migration component: selecting classifiers randomly (R), by numerosity (N), or by fitness (F)
$\delta_{\text{emigrants}}$	proportion of the classifiers that the XCS selects as emigrants
δ_{hold}	initial delay during which no migration occurs
$\mathcal{T}$	parameter of the topology module: unidirectional ring (U), bidirectional ring (B), or complete graph (C)
$\mathcal{D}$	parameter of deletion strategy: deletion by fitness (F), by action set size (A), or by prediction error (E)
(c, t)	parameters of the operator-allocation problem: number of available operators c and number of operators to be allocated t
action zero	special action indicating that no operator allocation is possible
$m(\cdot)$	multiplexer problem
$\overline{m}(\cdot)$	inverse multiplexer problem
k_{max}	maximum problem dimension
k_{avg}	equivalent problem dimension for complex problems
k_{cl}	problem dimension (schema order) of classifier
w_{cl}	problem dimension weight of a classifier when calculating equivalent problem dimension for complex problems
p_{explore}	subclause in calculating k_{avg} (Equation 4.7, page 61)
p_{exploit}	subclause in calculating k_{avg} (Equation 4.8, page 61)
$\text{rel}(\cdot)$	reliability function

Bibliography

Agarwal A, Zolotov V, and Blaauw DT. Statistical clock skew analysis considering intra-die process variations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(8):1231–1242. 2004.

Agarwal K, Deogun H, Sylvester D, and Nowka K. Power gating with multiple sleep modes. *International Symposium on Quality Electronic Design (ISQED)*, pp. 633–637. doi:10.1109/ISQED.2006.102. 2006.

Anguita D, Ridella S, and Rovetta S. Circuital implementation of support vector machines. *Electronics Letters*, 34(16):1596–1597. 1998.

Anguita D, Boni A, and Ridella S. A digital architecture for support vector machines: theory, algorithm, and FPGA implementation. *Transactions on Neural Networks*, 14(5):993–1009. 2003.

Appleby K, Fakhouri S, Fong L, Goldszmidt G, Kalantar M, Krishnakumar S, Pazel DP, Pershing J, and Rochwerger B. Oceano—SLA based management of a computing utility. *IEEE/IFIP International Symposium on Integrated Network Management*, pp. 855–868. doi:10.1109/INM.2001.918085. 2001.

Arndt G. *Verteiltes Learning Classifier System XCS*. Diploma thesis, University of Tübingen. 2010.

Ashby R. Principles of the self-organizing dynamic system. *Journal of General Psychology*, 37:125–128. 1947.

Ashby WR. Principles of the self-organizing system. Foerster HV and Zopf Jr GW (eds.) *Principles of Self-Organization: Transactions of the University of Illinois Symposium*. Pergamon. 1962.

Bar-Yam Y. A mathematical theory of strong emergence using multiscale variety. *Complexity*, 9(6):15–24. 2004.

Benini L, De Micheli G, Macii E, Poncino M, and Scarsi R. Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. *ACM Transactions on Design Automation of Electronic Systems*, 4:351–375. doi:10.1145/323480.323482. 1999.

Bernadó-Mansilla E and Garrell-Guiu JM. Accuracy-based learning classifier systems: models, analysis and applications to classification tasks. *Evolutionary Computation*, 11(3):209–238. 2003.

Bernauer A, Fritz D, and Rosenstiel W. Evaluation of the learning classifier system XCS for SoC run-time control. *Lecture Notes in Informatics*, volume 134, pp. 761–768. Gesellschaft für Informatik, Springer. 2008.

Bernauer A, Bringmann O, and Rosenstiel W. Generic self-adaptation to reduce design effort for system-on-chip. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 126–135. doi:10.1109/SASO.2009.41. 2009.

Bernauer A, Zeppenfeld J, Bringmann O, Herkersdorf A, and Rosenstiel W. Combining software and hardware LCS for lightweight on-chip learning. *Distributed Parallel and Biologically Inspired Systems, IFIP AICT 329*, pp. 279–290. Springer. doi:10.1007/978-3-642-15234-4_27. 2010.

Bernstein K, Frank DJ, Gattiker AE, Haensch W, Ji BL, Nassif SR, Nowak EJ, Pearson DJ, and Rohrer NJ. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4/5):433. 2006.

Bolchini C, Ferrandi P, Lanzi PL, and Salice F. Toward an FPGA implementation of XCS. *Congress on Evolutionary Computation*, volume 3, pp. 2053–2060. IEEE. 2005.

———. Evolving classifiers on field programmable gate arrays: Migrating XCS to FPGAs. *Journal of Systems Architecture*, 52(8-9):516–533. 2006.

Bonabeau E, Dorigo M, and Theraulaz G. *Swarm Intelligence - From Natural to Artificial Systems*. Oxford University, New York. 1999.

Borkar S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16. doi:10.1109/MM.2005.110. 2005.

———. Design perspectives on 22nm CMOS and beyond. *Design Automation Conference (DAC)*, pp. 93–94. 2009.

Borkar S, Karnik T, Narendra S, Tschanz J, Keshavarzi A, and De V. Parameter variations and impact on circuits and microarchitecture. *Design Automation Conference (DAC)*, pp. 338–342. ACM. doi:10.1145/775832.775920. 2003.

Brockmann W. Online machine learning for adaptive control. *IEEE International Workshop on Emerging Technologies and Factory Automation—Technology for the Intelligent Factory*, pp. 190–195. CRL, London. 1992.

Bull L, Studley M, Bagnall T, and Whittley I. On the use of rule-sharing in learning classifier system ensembles. *Congress on Evolutionary Computation*, pp. 612–617. IEEE. 2005.

Butts JA and Sohi GS. A static power model for architects. *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 191 –201. ACM. doi:10.1145/360128.360148. 2000.

Butz M and Wilson SW. An algorithmic description of XCS. Lanzi PL, Stolzmann W, and Wilson SW (eds.) *Advances in Learning Classifer Systems*, *Lecture Notes in Artificial Intelligence (LNAI)*, volume 1996, pp. 253–272. Springer, Berlin. doi: 10.1007/3-540-44640-0_15. 2001.

Butz MV. An implementation of the XCS classifier system in C. Technical Report 99021, The Illinois Genetic Algorithms Laboratory. 1999.

———. *Rule-Based Evolutionary Online Learning Systems.* Springer, Berlin. 2006.

Butz MV, Kovacs T, Lanzi PL, and Wilson SW. How XCS evolves accurate classifiers. Spector L et al. (eds.) *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 927–934. 2001.

Butz MV, Goldberg DE, and Tharakunnel K. Analysis and improvement of fitness exploitation in XCS: bounding models, tournament selection, and bilateral accuracy. *Evolutionary Computation*, 11(3):239–277. doi:10.1162/106365603322365298. 2003.

Butz MV, Goldberg DE, and Lanzi PL. Bounding learning time in XCS. IlliGAL report 2004003, University of Illinois at Urbana-Champaign, University of Illinois at Urbana-Champaign. 2004a.

Butz MV, Kovacs T, Lanzi PL, and Wilson SW. Toward a theory of generalization and learning in XCS. *IEEE Transactions on Evolutionary Computation*, 8(1). 2004b.

Butz MV, Lanzi PL, and Wilson SW. Hyper-ellipsoidal conditions in XCS: rotation, linear approximation, and solution structure. Cattolico M (ed.) *GECCO*, pp. 1457–1464. ACM. doi:10.1145/1143997.1144237. 2006.

Bäck T, Fogel DB, and Michalewicz T (eds.) *Evolutionary Computation 1.* Institue of Physics, Bristol, U. K. 2002.

Cai L and Gaijski D. Transaction level modeling: An overview. *Conference on Hardware–Software Codesign and System Synthesis (CODES+ISSS)*, pp. 19–24. 2003.

Correale A Jr. Overview of the power minimization techniques employed in the IBM PowerPC 4xx embedded controllers. *International Symposium on Low Power Design*, pp. 75–80. ACM. doi:10.1145/224081.224095. 1995.

Dam H, Rojanavasu P, Abbass H, and Lokan C. Distributed learning classifier systems. *Learning Classifier Systems in Data Mining*, pp. 69–91. 2008.

Dam HH, Abbass HA, and Lokan C. DXCS: an XCS system for distributed data mining. Rothlauf F (ed.) *GECCO*, pp. 1883–1890. ACM. doi:10.1145/1068009.1068326. 2005.

Degalahal V, Li L, Narayanan V, Kandemir M, and Irwin MJ. Soft errors issues in low-power caches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1157 – 1166. doi:10.1109/TVLSI.2005.859474. 2005.

Dias FM, Antunes A, and Mota AM. Artificial neural networks: a review of commercial hardware. *Engineering Applications of Artificial Intelligence*, 17(8):945–952. 2004.

Dittrich P. The bio-chemical information processing metaphor as a programming paradigm for organic computing. Brinkschulte U, Becker J, Fey D, Hochberger C, Martinetz T, Müller-Schloer C, Schmeck H, Ungerer T, and Würtz RP (eds.) *Architecture of Computing Systems (ARCS) Workshops*, pp. 95–99. VDE, Innsbruck, Austria. 2005.

Dixon PW, Corne DW, and Oates MJ. A ruleset reduction algorithm for the XCS learning classifier system. Lanzi PL, Stolzmann W, and Wilson SW (eds.) *Learning Classifer Systems*, *Lecture Notes in Artificial Intelligence (LNAI)*, volume 2661, pp. 20–29. Springer, Berlin. 2003.

Doriga M and Schnepf U. Genetics-based machine learning and behavior-based robotics: A new synthesis. *Transactions on Systems, Man and Cybernetics*, 23(1):141–154. 1993.

Dorigo M. ALECSYS and the AutonoMouse: Learning to control a real robot by distributed classifier systems. *Machine Learning*, 19(3):209–240. doi:10.1023/A: 1022649410928. 1995.

Efraimidis PS and Spirakis PG. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185. doi:10.1016/j.ipl.2005.11.003. 2006.

El Sayed Auf A, Mösch F, and Litza M. How the six-legged walking machine OSCAR handles leg amputations. Published on CD, Article available at `http://www.iti.uni-luebeck.de/fileadmin/user_upload/Papers/ElSayedAuf%_etal_06.pdf` (retrieved January 10, 2011). From Animals to Animats 9: Ninth International Conference on the Simulation of Adaptive Behavior (SAB). 2006.

Ernst D, Kim NS, Das S, Pant S, Pham T, Rao R, Ziesler C, Blaauw D, Austin T, Mudge T, and Flautner K. Razor: A low-power pipeline based on circuit-level timing speculation. *Proc. 36th Intl. Symp. Microarch.*, pp. 7–18. 2003.

Fetzer ES. Using adaptive circuits to mitigate process variations in a microprocessor design. *IEEE Design and Test of Computers*, 23:476–483. doi:10.1109/MDT.2006.159. 2006.

Fey D and Schmidt D. Marching-pixels: a new organic computing paradigm for smart sensor processor arrays. *Conference on Computing Frontiers (CF)*, pp. 1–9. ACM. doi:10.1145/1062261.1062264. 2005.

Fredivianus N, Prothmann H, and Schmeck H. XCS revisited: A novel discovery component for the eXtended classifier system. Deb K, Bhattacharya A, Chakraborti N, Chakroborty P, Das S, Dutta J, Gupta S, Jain A, Aggarwal V, Branke J, Louis S, and Tan K (eds.) *Simulated Evolution and Learning*, *Lecture Notes in Computer Science (LNCS)*, volume 6457, pp. 289–298. Springer. doi:10.1007/978-3-642-17298-4_30. 2010.

Fromm J. Types and forms of emergence. `http://arxiv.org/abs/nlin.AO/0506028`. 2005.

Genossar D and Shamir N. Intel Pentium M processor power estimation, budgeting, optimization, and validation. *Intel Technology Journal*, 7(2):44–49. 2003.

Gershenson C and Heylighen F. When can we call a system self-organizing? Banzhaf W, Christaller T, Dittrich P, Kim JT, and Ziegler J (eds.) *Advances in Artificial Life*, number 2801 in Lecture Notes in Artificial Intelligence (LNAI), pp. 606–614. Springer, Dortmund, Germany. 2003.

Gershoff M and Schulenburg S. Collective behavior based hierarchical XCS. *Conference Companion on Genetic and Evolutionary Computation*, pp. 2695–2700. ACM. 2007.

Golda A and Kos A. Temperature influence on power consumption and time delay. *Euromicro Symposium on Digital Systems Design*, pp. 378–382. IEEE Computer Society. doi:10.1109/DSD.2003.1231970. 2003.

Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison–Wesley, Boston, MA, USA. 1989.

Guerin S and Kunkle D. Emergence of constraint in self-organizing systems. *Nonlinear Dynamics, Psychology, and Life Sciences*, 8(2):131–146. 2004.

Gutnik V and Chandrakasan AP. Embedded power supply for low-power DSP. *IEEE Transactions on Very Large Scale Integration Systems*, 5:425–435. doi:10.1109/92.645069. 1997.

Herrmann K, Werner M, and Mühl G. A methodology for classifying self-organizing software systems. *International Transactions on Systems Science and Applications*, 2(1):41–50. 2006.

Heylighen F. Modelling emergence. *World Futures: the Journal of General Evolution*, 31:89–104. 1991.

———. The science of self-organization and adaptivity. Kiel LD (ed.) *Knowledge Management, Organizational Intelligence and Learning, and Complexity*, Encyclopedia of Life Support Systems (EOLSS). Eolss, Oxford, UK. 2004.

Holland JH. Adaptation. Rosen R and Snell FM (eds.) *Progress in Theoretical Biology*, pp. 263–293. Academic, New York, NY, USA. 1976.

Holland JH and Reitman JS. Cognitive systems based on adaptive algorithms. Waterman DA and Hayes-Roth F (eds.) *Pattern-directed inference systems*. Academic, New York, NY, USA. 1978.

Holland JH, Booker LB, Colombetti M, Dorigo M, Goldberg DE, Forrest S, Riolo RL, Smith RE, Lanzi PL, Stolzmann W, and Wilson SW. What is a learning classifier system? Lanzi et al. (2000), pp. 3–32. doi:10.1007/3-540-45027-0_1. 2000.

Huang W, Stan MR, Skadron K, Sankaranarayanan K, Ghosh S, and Velusam S. Compact thermal modeling for temperature-aware design. *Design Automation Conference (DAC)*, pp. 878–883. ACM. 2004.

Hughes CJ, Srinivasan J, and Adve SV. Saving energy with architectural and frequency adaptations for multimedia applications. *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 250–261. IEEE Computer Society. doi:10.1109/MICRO.2001.991123. 2001.

Hurst AP. Automatic synthesis of clock gating logic with controlled netlist perturbation. *Design Automation Conference (DAC)*, pp. 654–657. ACM. doi:10.1145/1391469.1391637. 2008.

Hurst J, Bull L, and Melhuish C. TCS learning classifier system controller on a real robot. Guervós J, Adamidis P, Beyer HG, Schwefel HP, and Fernández-Villacañas JL (eds.) *Parallel Problem Solving from Nature (PPSN)*, *Lecture Notes in Computer Science (LNCS)*, volume 2439, pp. 588–597. Springer. doi:10.1007/3-540-45712-7_57. 2002.

International Roadmap Committee. International technology roadmap for semiconductors. http://www.itrs.net/reports.html. 2008.

Irick K, DeBole M, Narayanan V, and Gayasen A. A hardware efficient support vector machine architecture for FPGA. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 304–305. IEEE Computer Society, Washington, DC, USA. doi:10.1109/FCCM.2008.40. 2008.

Jayaseelan R and Mitra T. Dynamic thermal management via architectural adaptation. *Design Automation Conference (DAC)*, pp. 484–489. ACM. doi:10.1145/1629911.1630038. 2009.

Kapadia H, Benini L, and De Micheli G. Reducing switching activity on datapath buses with control-signal gating. *IEEE Journal of Solid-State Circuits*, 34(3):405–414. doi:10.1109/4.748193. 1999.

Karl E, Sylvester D, and Blaauw D. Timing error correction techniques for voltage-scalable on-chip memories. *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pp. 3563–3566. doi:10.1109/ISCAS.2005.1465399. 2005.

Kawaguchi H, Zhang G, Lee S, Shin Y, and Sakurai T. A controller LSI for realizing VDD-hopping scheme with off-the-shelf processors and its application to MPEG4 system. *IEICE Transactions on Electronics*, E85-C(2):263–271. 2002.

Kephart JO and Chess DM. The vision of autonomic computing. *Computer*, 36(1):41–50. 2003.

Kovacs T. XCS classifier system reliably evolves accurate, complete, and minimal representations for boolean functions. Roy, Chawdhry, and Pant (eds.) *Soft Computing in Engineering Design and Manufacturing*, pp. 59–68. Springer. 1997.
URL `http://hdl.handle.net/10068/653236`

Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Wells C, and Zhao B. Oceanstore: an architecture for global-scale persistent storage. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 190–201. ACM. doi:10.1145/378993.379239. 2000.

Landsberg PT. *Thermodynamics and statistical mechanics.* Oxford University, London. 1978.

———. Can entropy and "order" increase together? *Physics Letters*, 102A(4):171–173. 1984.

Lanzi PL, Stolzmann W, and Wilson SW (eds.) *Learning Classifier Systems: From Foundations to Applications*, *Lecture Notes in Artificial Intelligence (LNAI)*, volume 1813. Springer, Berlin. doi:10.1007/3-540-45027-0. 2000.

Lendaris GG. On the definition of self-organizing systems. *Proceedings of the IEEE.* 1964.

Lipsa G, Herkersdorf A, Rosenstiel W, Bringmann O, and Stechele W. Towards a framework and a design methodology for autonomic SoC. *IEEE International Conference on Autonomic Computing (ICAC).* Seattle, USA. 2005.

Lohman GM and Lightstone SS. SMART: making DB2 (more) autonomic. *International Conference on Very Large Data Bases (VLDB)*, pp. 877–879. VLDB Endowment. 2002. URL `http://portal.acm.org/citation.cfm?id=1287369.1287444`

Matsumaru N and Dittrich P. Organization-oriented chemical programming for the organic design of distributed computing systems. *International Conference on Bio Inspired Models of Network, Information and Computing Systems (BIONETICS).* ACM. doi:10.1145/1315843.1315861. 2006.

McCaffrey J. Generating the mth lexicographical element of a mathematical combination. `http://msdn.microsoft.com/en-us/library/aa289166%28VS.71%29.aspx`. 2005.

Melhem R, Mossé D, and Elnozahy EM. The interplay of power management and fault recovery in real-time systems. *IEEE Transactions on Computers*, 53:217–231. doi:10.1109/TC.2004.1261830. 2004.

Mishra B, Al-Hashimi BM, and Zwolinski M. Variation resilient adaptive controller for subthreshold circuits. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 142–147. IEEE. 2009.

Mutoh S, Douseki T, Matsuya Y, Aoki T, Shigematsu S, and Yamada J. 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8):847–854. doi:10.1109/4.400426. 1995.

Mösch F, Litza M, Auf A, Maehle E, Großpietsch K, and Brockmann W. ORCA—towards an organic robotic control architecture. de Meer H and Sterbenz J (eds.) *Self-Organizing Systems*, *Lecture Notes in Computer Science (LNCS)*, volume 4124, pp. 251–253. Springer. doi:10.1007/11822035_24. 2006.

Müller-Schloer C. Organic computing: on the feasibility of controlled emergence. *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 2–5. ACM. doi:10.1145/1016720.1016724. 2004.

Müller-Schloer C, von der Malsburg C, and Würtz RP. Organic computing. *Informatik Spektrum*, 27(4):332–336. doi:10.1007/s00287-004-0409-6. 2004.

Münch M, Wurth B, Mehra R, Sproch J, and Wehn N. Automating RT-level operand isolation to minimize power consumption in datapaths. *Design, Automation and Test in Europe (DATE)*, pp. 624–633. ACM. doi:10.1145/343647.343873. 2000.

Naffziger S, Stackhouse B, Grutkowski T, Josephson D, Desai J, Alon E, and Horowitz M. The implementation of a 2-core, multi-threaded itanium family processor. *Solid-State Circuits*, 41(1):197–209. 2005.

Narayanan V and Xie Y. Reliability concerns in embedded system designs. *Computer*, 39(1):118–120. doi:10.1109/MC.2006.31. 2006.

Nicolaidis M. Time redundancy based soft-error tolerance to rescue nanometer technologies. *IEEE VLSI Test Symposium*, pp. 86–94. 1999.

Parashar M and Hariri S. Autonomic computing: An overview. Banâtre JP, Fradet P, Giavitto JL, and Michel O (eds.) *Unconventional Programming Paradigms*, *Lecture Notes in Computer Science (LNCS)*, volume 3566, pp. 257–269. Springer. doi:10.1007/11527800_20. 2005.

Parunak HVD and Brueckner S. Entropy and self-organization in multi-agent systems. *International Conference on Autonomous Agents*, pp. 124–130. 2001.

Pham D, Asano S, Bolliger M, Day MN, Hofstee HP, Johns C, Kahle J, Kameyama A, Keaty J, Masubuchi Y, Riley M, Shippy D, Stasiak D, Suzuoki M, Wang M, Warnock J, Weitzel S, Wendel D, Yamazaki T, and Yazawa K. The design and implementation of a first-generation CELL processor. *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference (ISSCC)*, volume 1, pp. 184–185, 592. doi:10.1109/ISSCC.2005.1493930. 2005a.

———. The design and implementation of a first-generation CELL processor – a multi-core SoC. *International Conference on Integrated Circuit Design and Technology (ICICDT)*, pp. 49–52. doi:10.1109/ICICDT.2005.1502588. 2005b.

Pham D, Behnen E, Bolliger M, Hofstee HP, Johns C, Kahle J, Kameyama A, Keaty J, Le B, Masubuchi Y, Posluszny S, Riley M, Suzuoki M, Wang M, Warnock J, Weitzel S, Wendel D, and Yazawa K. The design methodology and implementation of a first-generation CELL processor: A multi-core SoC. *IEEE Custom Integrated Circuits Conference*, pp. 45–49. doi:10.1109/CICC.2005.1568604. 2005c.

Pham D, Aipperspach T, Boerstler D, Bolliger M, Chaudhry R, Cox D, Harvey P, Harvey PM, Hofstee HP, Johns C, Kahle J, Kameyama A, Keaty J, Masubuchi Y, Pham M, Pille J, Posluszny S, Riley M, Stasiak DL, Suzuoki M, Takahashi O, Warnock J, Weitzel S, Wendel D, and Yazawa K. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits*, 41(1):179 – 196. doi:10.1109/JSSC.2005.859896. 2006.

Prothmann H, Rochner F, Tomforde S, Branke J, Müller-Schloer C, and Schmeck H. Organic control of traffic lights. Rong C, Jaatun M, Sandnes F, Yang L, and Ma J (eds.) *Autonomic and Trusted Computing*, *Lecture Notes in Computer Science (LNCS)*, volume 5060, pp. 219–233. Springer. doi:10.1007/978-3-540-69295-9_19. 2008.

Rabaey JM and Malik S. Challenges and solutions for late- and post-silicon design. *IEEE Design and Test of Computers*, 25:296–302. doi:10.1109/MDT.2008.91. 2008.

Rakitsch B, Bernauer A, Bringmann O, and Rosenstiel W. Pruning population size in XCS for complex problems. *World Congress on Computational Intelligence (WCCI)*, pp. 3383–3390. IEEE, Barcelona, Spain. 2010.

Richter U, Mnif M, Branke J, Müller-Schloer C, and Schmeck H. Towards a generic observer/controller architecture for organic computing. INFORMATIK *2006, Informatik für Menschen, Band 1, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e. V. (GI)*, pp. 112–119. 2006.

Richter U, Prothmann H, and Schmeck H. Improving XCS performance by distribution. Li X, Kirley M, Zhang M, Green D, Ciesielski V, Abbass H, Michalewicz Z, Hendtlass T, Deb K, Tan K, Branke J, and Shi Y (eds.) *Simulated Evolution and Learning*, *Lecture Notes in Computer Science (LNCS)*, volume 5361, pp. 111–120. Springer. doi:10.1007/978-3-540-89694-4_12. 2008.

Rochner F, Prothmann H, Branke J, Müller-Schloer C, and Schmeck H. An organic architecture for traffic light controllers. Christian Hochberger RL (ed.) INFORMATIK *2006, Informatik für Menschen, Band 1, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e. V. (GI)*, *Lecture Notes in Informatics (LNI)*, volume P-93, pp. 120–127. Dresden. 2006.

Sander B, Bernauer A, and Rosenstiel W. Design and run-time reliability at the electronic system level. *IPSJ Transactions on System LSI Design Methodology*, volume 3, pp. 140–160. 2010.

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

Schmeck H. Organic computing—a new vision for distributed embedded systems. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 201–203. doi:10.1109/ISORC.2005.42. 2005.

Schmeck H, Müller-Schloer C, Çakar E, Mnif M, and Richter U. Adaptivity and self-organization in organic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 5:10:1–10:32. doi:10.1145/1837909.1837911. 2010.

Schumacher C, Leupers R, Petras D, and Hoffmann A. parsc: synchronous parallel SystemC simulation on multi-core host architectures. *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS, pp. 241–246. ACM. doi:10.1145/1878961.1879005. 2010.

Schweizer T, Oliveira J, Kuhn T, and Rosenstiel W. Charge recycling in voltage-dithered circuits. *Journal of Low Power Electronics*, 6(2):291–299. 2010.

Skadron K, Stan MR, Huang W, Velusamy S, Sankaranarayanan K, and Tarjan D. Temperature-aware microarchitecture. SIGARCH *Computer Architecture News*, 31(2):2–13. doi:10.1145/871656.859620. 2003.

Skadron K, Stan MR, Sankaranarayanan K, Huang W, Velusamy S, and Tarjan D. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimizations*, 1(1):94–125. doi:10.1145/980152.980157. 2004.

Skinner BT, Nguyen HT, and Liu DK. Distributed classifier migration in XCS for classification of electroencephalographic signals. *IEEE Congress on Evolutionary Computation*, pp. 2829–2836. 2007.

Smith RE, Dike BA, Ravichandran B, El-Fallah A, and Mehra RK. The fighter aircraft LCS: A case of different LCS goals and techniques. Lanzi et al. (2000), pp. 283–300. doi:10.1007/3-540-45027-0_15. 2000.

Smith SF. Flexible learning of problem solving heuristics through adaptive search. *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 422–425. Morgan Kaufmann, San Francisco, CA, USA. 1983.

Studley M. Learning classifier systems for multi-objective robot control. Technical Report UWELCSG06-005, University of the West of England, LCS Group, Bristol BS16 1QY, U. K. 2006.

Studley M and Bull L. X-TCS: accuracy-based learning classifier system robotics. *Evolutionary Computation*, volume 3, pp. 2099–2106. doi:10.1109/CEC.2005.1554954. 2005.

Sutton RS. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44. 1988.

SystemC. IEEE standard SystemC language reference manual. IEEE Std 1666-2005. 2005.

Tarasewich P and McMullen PR. Swarm intelligence. *Communications of the ACM*, 45(8):62–67. 2002.

Tiwari A, Sarangi SR, and Torrellas J. Recycle: pipeline adaptation to tolerate process variation. *International Symposium on Computer Architecture (ISCA)*, pp. 323–334. ACM. doi:10.1145/1250662.1250703. 2007.

Tiwari V, Malik S, and Ashar P. Guarded evaluation: pushing power management to logic synthesis/design. *International Symposium on Low Power Design*, pp. 221–226. ACM. doi:10.1145/224081.224120. 1995.

Velusamy S, Huang W, Lach J, and Stan M. Monitoring temperature in FPGA based SoCs. *IEEE International Conference on Computer Design (ICCD)*. doi:10.1109/ICCD.2005.78. 2005.

Venturini G. Adaptation in dynamic environments through a minimal probability of exploration. *International Conference on Simulation of Adaptive Behavior: from animals to animats 3 (SAB)*, pp. 371–379. MIT, Cambridge, MA, USA. 1994.

Vitter JS. Random sampling with a reservoir. *ACM Transacations on Mathematical Software*, 11:37–57. doi:10.1145/3147.3165. 1985.

Watkins CJCH. *Learning from Delayed Rewards*. Ph.D. thesis, King's College. 1989.

Watkins CJCH and Dayan P. Q-learning. *Machine Learning*, 8:279–292. 1992.

Wegener A, Schiller E, Hellbrück H, Fekete S, and Fischer S. Hovering data clouds: A decentralized and self-organizing information system. de Meer H and Sterbenz J (eds.) *Self-Organizing Systems*, *Lecture Notes in Computer Science (LNCS)*, volume 4124, pp. 243–247. Springer. doi:10.1007/11822035_22. 2006.

Weste N and Eshraghian K. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison–Wesley, 2nd edition. 1993.

Whitley D and Kauth J. GENITOR: a different genetic algorithm. *Rocky Mountain Conference on Artificial Intelligence*, pp. 118–130. Denver, CO, USA. 1988.

Widrow B, Rumelhart DE, and Lehr MA. Neural networks: applications in industry, business and science. *Communications of the ACM*, 37(3):93–105. doi:10.1145/175247.175257. 1994.

Wilson SW. Classifier systems and the animat problem. *Machine Learning*, 2(3):199–228. doi:10.1007/BF00058679. 1987.

———. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18. doi:10.1162/evco.1994.2.1.1. 1994.

———. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175. doi:10.1162/evco.1995.3.2.149. 1995.

———. Generalization in the XCS classifier system. Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, and Riolo R (eds.) *Conference on Genetic Programming*, pp. 665–674. Morgan Kaufmann. 1998.

———. Compact rulesets from XCSI. Lanzi PL, Stolzmann W, and Wilson SW (eds.) *Advances in Learning Classifier Systems*, *Lecture Notes in Artificial Intelligence (LNAI)*, volume 2321, pp. 196–208. Springer. 2002.

Winfield AFT and Holland OE. The application of wireless local area network technology to the control of mobile robots. *Microprocessors and Microsystems*, 23(10):597–607. 2000.

Zeppenfeld J and Herkersdorf A. Autonomic workload management for multi-core processor systems. *International Conference on Architecture of Computing Systems (ARCS)*, pp. 49–60. Springer, Hannover, Germany. 2010.

Zeppenfeld J, Bouajila A, Stechele W, and Herkersdorf A. Learning classifier tables for autonomic systems on chip. Hegering HG, Lehmann A, Ohlbach HJ, and Scheideler C (eds.) *GI Jahrestagung (2)*, *Lecture Notes in Informatics (LNI)*, volume 134, pp. 771–778. GI. 2008.

Zhang Y and Chakrabarty K. Energy-aware adaptive checkpointing in embedded real-time systems. *Conference on Design, Automation and Test in Europe - Volume 1 (DATE)*, pp. 918–923. IEEE Computer Society. doi:10.1109/DATE.2003.10177. 2003.

Zhu D. Reliability-aware dynamic energy management in dependable embedded real-time systems. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 397–407. IEEE Computer Society. doi:10.1109/RTAS.2006.36. 2006.

Zhu D and Aydin H. Reliability-aware energy management for periodic real-time tasks. *IEEE Transactions on Computers*, 58(10):1382–1397. doi:10.1109/TC.2009.56. 2009.

[this page intentionally left blank]

[this page intentionally left blank]

[this page intentionally left blank]

[this page intentionally left blank]