Learning Classifier Tables for Autonomic Systems on Chip

J. Zeppenfeld, A. Bouajila, W. Stechele, A. Herkersdorf
Institute for Integrated Systems
Technische Universität München

Abstract: This paper introduces a new hardware-based machine learning building block – called Learning Classifier Table (LCT) – for the run-time reliability, performance and power optimization of future generations of Systems-on-Chip. LCT inherits concepts from the reinforcement learning techniques found in Learning Classifier Systems. Prediction weighted LCT rule evaluation is implemented on a clock cycle scale with low hardware complexity.

1 Introduction

The Autonomic Systems on Chip (ASoC [1]) proposal envisions future nanometer VLSI components that autonomously cope with sporadic logic and timing errors induced by natural radiation and technology- or environmentally-related variations. A self-correcting CPU pipeline technique was proposed in [2] to detect and immediately correct SET/SEU errors "on the fly". The goal of our present work is to add learning capabilities to this self-healing soft-core CPU in order to allow for a variety of situation-dependent countermeasures to error occurrences under different SoC operating conditions. We investigated the use of Learning Classifier Systems (LCS [3], XCS [4]) for combined reliability, power and performance (RPP) optimization of SoCs.

LCS is a reinforcement machine learning technique that attempts to maximize long term rewards for actions performed in response to specific situations. While LCS is typically implemented in software at a fairly high level [6] or utilizes a majority of hardware resources [7], this paper presents a fast and resource efficient hardware alternative called Learning Classifier Table (LCT) which, although not equivalent to existing LCS systems, draws heavily on their concepts. Rules are selected from a classifier population based on their condition matching the current input of monitor data, forming a so-called match set. From this match set a single rule is selected based on its fitness, which in LCT corresponds to the rule's prediction of the expected reward rather than its accuracy as in XCS. The action of the selected rule is carried out, followed by an update of the rule's prediction when a reward is returned by the system. In order to keep the LCT hardware implementation as compact and dependable as possible, other XCS features such as prediction error calculations, action set creation and the execution of genetic operators are not implemented in hardware. Instead, software extensions are anticipated to provide at least some of these missing components.

The remainder of this paper is organized as follows: Section 2 expands on the design above, giving a detailed overview of the fundamental LCT structure. Section 3 explores the proposed algorithm for fitness weighted rule selection, taking into account specific opportunities and limitations of the targeted hardware implementation. Preliminary synthesis and simulation results of this central rule selection algorithm are presented as well. Section 4 examines critical timings associated with rule matching, action execution and updates of the fitness value. The paper then concludes with Section 5.

2 Classifier System Overview

The embedding of the proposed LCT system among ordinary SoC macros is presented in Figure 1. The functional element (FE) shown at the bottom represents a typical hardware IP, such as a CPU core, an I/O peripheral, on-chip memory or hardware accelerator. Specialized monitors track FE reliability, performance and power metrics such as transient error rate, FE utilization and die temperature. Monitor signals are aggregated and thresholded to obtain a more compact representation for evaluation by the LCT rule base. For example, occasional timing errors may be acceptable if they can be corrected at the device level [2], but recurring timing errors may reveal a defective core, requiring it to reduce its frequency or shut down. Rather than evaluating every error occurrence individually, a counter can be used to aggregate errors over a certain time interval. Different counter thresholds correspond to different ranges of error rate. Passing information to the LCT only on which thresholds have been exceeded allows the evaluator to deal with errors in groups rather than individually.

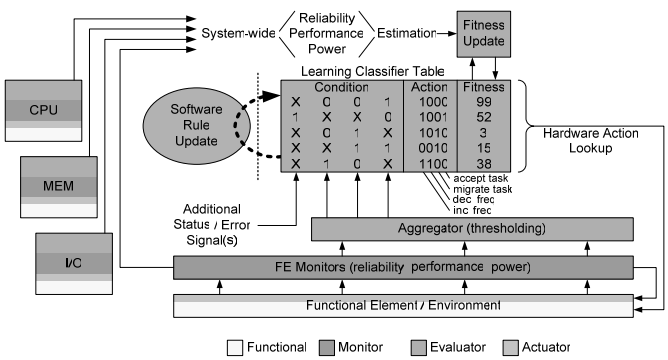

Figure 1: Overview of an LCT embedded in a system

The LCT determines an appropriate action to be taken based on the incoming monitor aggregates, and will forward this action to the FE for execution. Possible actions for a CPU FE include an increase or decrease in frequency/supply voltage, requests for a task migration, or the en-/disabling of certain subcomponents, e.g. the floating point unit of the processor core. The LCT implements a simplified LCS which is optimized for hardware implementation. Similarly to other classifier systems, an LCT contains a list of rules that make up the classifier population. Each rule consists of a condition, an action, and a fitness value. Given the monitor and actuator signals listed above, a typical rule

may for example state: *If* the CPU utilization is high *and* the number of errors is low (condition), *then* increase the CPU frequency (action).

In LCS, all matching rules are combined into what is referred to as the match set, each rule of which will generally propose a different action. The preferred action is selected from the match set based on the fitness value of the proposing rule. Whereas the fitness value of XCS reflects the accuracy of the rule's reward prediction, for simplicity the rules in our LCT will directly use the predicted reward as their fitness, similar to XCS' predecessor ZCS [5]. This makes calculating a prediction array based on the matching rules unnecessary.

Numerous methods have been suggested in the literature for selecting a single rule or action from the match set based on its predicted reward. These include always selecting the action with the highest reward prediction, random selection with individual probabilities weighted by their prediction, or completely random selection [4]. Although selecting the rule with the highest prediction may appear to best utilize the momentary knowledge of the classifier, it cannot guarantee to achieve a global optimum in the RPP space of the SoC. The opposite extreme, completely random selection, disregards predictions entirely, and does not make use of learning. In order to balance the exploitation of existing knowledge stored in the classifier and the exploration of new or previously untested rules, we employ a randomized selection scheme where the selection probability of each rule is weighted by its prediction (commonly known as roulette-wheel selection). Rules with a high prediction have a larger chance of selection, but do not entirely bar the selection of even those rules with a very low prediction.

Dynamic fitness and prediction updates are one important part of learning within an LCS/XCS. Another important aspect is the genetic algorithm (GA) or the genetic operators used to modify the population of classifiers over time [3]. Although genetic operators such as cross over and mutation can be readily implemented in hardware, choosing appropriate rules to be crossed or replaced (selection) can be difficult. More importantly, a hardware system must avoid generating rules with unpredictable results at run time, which is difficult to guarantee across essentially random genetic operators. Our LCT implementation will therefore limit its learning abilities to run-time prediction updates, but will allow for learning algorithms, such as a GA, to be implemented in software. This is made possible by mapping the LCT's rules into the system's address space, allowing the rules to be dynamically accessed and modified.

3 Rule Selection Algorithm

3.1. Fitness Weighting

The pseudo code in Table 1 shows the algorithm used for efficient randomized rule selection in hardware, the novelty of which lies in the fact that it includes both the match set "creation" and roulette wheel selection of an appropriate rule from within the classifier population, without the need to first generate an explicit match set by copying matching rules to an intermediary memory. The algorithm is based on single-pass

weighted reservoir sampling [8][9], with the special case of a reservoir size of one, since only a single rule will be selected for execution. The core concept of the algorithm is to keep track of the total prediction (fitness) sum of any already matched rules, and to weight the fitness of further matching rules against the fitness sum seen so far. This basically translates into a probability for selecting a new matching rule (over the previously selected rule) of the new rule's fitness against the sum of all matching rule's fitnesses seen so far, or $P(n) = F_n/\Sigma F_n$, where F_n is the new rule's fitness and ΣF_n is the sum of the fitnesses of all matching rules seen so far.

```
1:   ΣF = 0
2:   LOOP over all rules
3:     IF rule matches
4:       ΣF += rule fitness
5:       IF Random % ΣF < rule fitness
6:         act = rule action
7:     END IF
8:   END LOOP
9:   Perform act
```

Table 1: Fitness-weighted rule selection algorithm

Using this algorithm, the first matching rule will always be selected, since it has a selection probability $P(1) = F_1/(F_1) = 1$. Note, however, that the action associated with this rule is not yet executed, but is instead stored as the action to take *iff* no later rule's action is selected for execution. The second matching rule has a selection probability $P(2) = F_2/(F_1+F_2)$ over the first rule, which exactly corresponds to the weighted selection of one of two options. Things become a little more interesting with the selection weighting of the third matching rule, which has a probability $P(3) = F_3/(F_1+F_2+F_3)$ for being selected over the previous two rules, leaving a probability of $P(1||2) = 1-P(3) = (F_1+F_2)/(F_1+F_2+F_3)$ for keeping the previously selected rule. Since the second rule was chosen over the first with probability P(2), this results in an updated probability $P'(2) = P(1||2)\cdot P(2) = F_2/(F_1+F_2+F_3)$ for selecting the second and $P'(1) = P(1||2)\cdot(1-P(2)) = F_1/(F_1+F_2+F_3)$ for selecting the first rule, just as we would expect from a fitness weighted probability across all three rules. Additional matching rules will influence the probabilities of the previous ones in a similar manner, with each of the N total matching rules in the classifier population eventually having a selection probability

$$P(i) = F_i / \sum_{n=1}^{N} F_n \ . \qquad (1)$$

3.2. Efficient LCT Hardware Implementation in FPGA

Preliminary synthesis results of the LCT rule selection algorithm in Xilinx Virtex-II FPGA technology reveal less than 5% overhead compared to a standard Leon 2 processor core, as shown in Table 2. Note that this figure only reflects the rule selection algorithm, including neither the fitness update logic nor the cycle delay timers discussed in Section 4, which we do not however expect to cause a significant increase in the numbers presented here.

	HW LCT	Leon 2	Overhead
Slices	38	2979	1.3 %
FFs / LUTs	45 / 59	1591 / 5450	< 3 %
BRAMs	1	6	17 %
Multipliers	1	0	-
Period	7.6 ns	24.5 ns	-
Frequency	131 MHz	42.6 MHz	-

Table 2: LCT synthesis results on Xilinx Virtex-2 Pro (XC2VP30-6)

Although the intended target architecture of the ASoC project is ASIC, we extensively use Xilinx FPGAs for prototyping, and therefore base our classifier table dimensions on the resources available in FPGA technology. A single Virtex-II block RAM (BRAM) allows for a maximum of 512 rules composed of 16 bits for the condition, 8 bits for the action, and 12 bits for the fitness value and various timing flags discussed in Section 4. Iterating (lines 2-8 of the algorithm in Table 1) over such an embedded memory in hardware can be accomplished with a finite state machine. Comparing the monitor signal with the rule's condition (line 3) and incrementing the fitness sum on a match (line 4) are both trivial operations. More difficult is the decision of whether to choose the currently matching rule over a previously selected rule (line 5), since this requires a modulo (division) operation, which is very costly in hardware. By rearranging terms it is fortunately possible to convert the modulo division into a multiplication (using one Virtex-II multiplier macro). While a multiplication is still fairly expensive in terms of area and power overheads, it is much less costly than a division. The pseudorandom numbers required to randomize rule selection can be generated fairly easily in hardware using linear feedback shift registers (LFSR) [10].

While stepping through the rules table, it is necessary to both read and write each rule to compare the rule against the current monitor value and update e.g. its prediction value. Hence, at least two clock cycles per individual rule lookup are required, resulting in a total lookup delay of approximately 10 µs for a maximum-sized table with 512 rules running at 100 MHz. Adding an additional delay of 10 µs to allow for the action's effects to percolate through the system still allows for several ten-thousand lookups and actions per second, which should easily suffice for the purpose of runtime RPP optimization.

3.3. Initial LCT Simulation Results

Although no hardware testing of the presented LCT implementation has been performed, initial simulations of an equivalent software model demonstrate promising results. The results shown in Figure 2 were obtained from a simulated system composed of two processor cores, a shared bus and memory system, and an I/O interface providing a constant workload that is distributed equally between the processors. One of the processor cores is periodically toggled on and off, resulting in a changing workload to the other core, as can be seen from the periodic utilization spikes in the figure. An LCT connected to the constantly functioning core adjusts the core's frequency to keep the utilization near a predetermined level of fifty percent. Initially, the LCT requires more time to properly adjust the frequency, resulting in regular utilization spikes reaching 100 percent. To-

wards the end of the simulation, the LCT has learned to adjust the frequency more rapidly, and is usually able to catch utilization spikes before they reach the maximum. The frequency adjustments towards the end of simulation are also more coarse-grained, indicating that the LCT has learned that in some cases performing no frequency adjustment may actually be the preferable action.

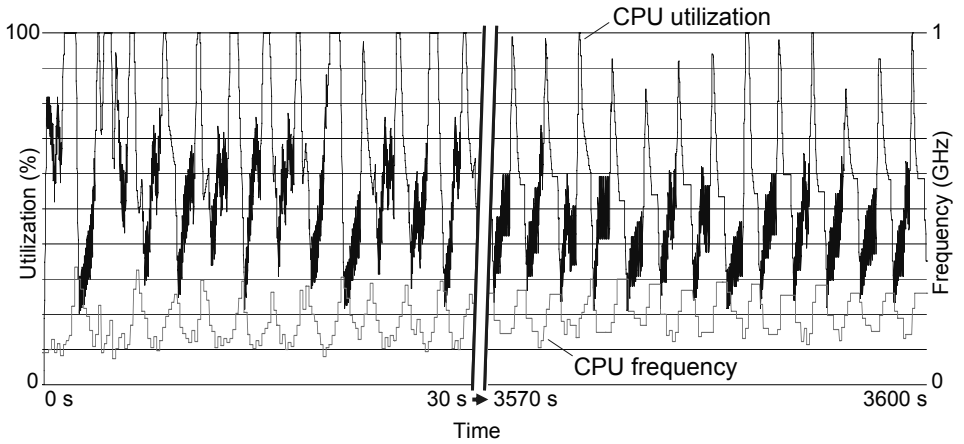

Figure 2: LCT simulation results

4 LCT Cycle Timing

As shown in Figure 3, every classifier lookup is associated with two delays. The first, the lookup delay (T_{LU}), measures the time between the sampling of the current monitor signals and the execution of an appropriate action. This corresponds to the time taken to traverse the rule table and to decide on an appropriate action (or no action) to be taken. Recall from the previous section that the action is not executed until the entire table has been traversed. Once an action has been selected, it is necessary to wait for a certain amount of time until the effects of that action have percolated through the system. We will refer to this delay between the action execution of one lookup cycle and the monitor sampling of the following cycle as the action delay (T_{Act}). Together, the lookup delay and action delay form the cycle period (T_P), which determines the overall LCT lookup frequency.

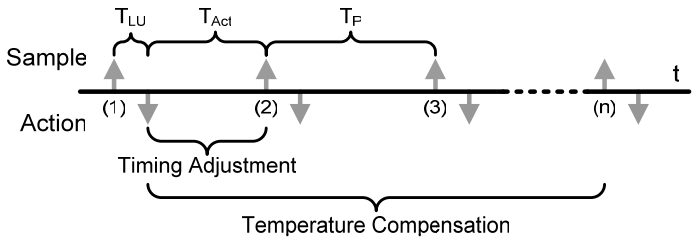

Figure 3: LCT Cycle Timing

Since the fitness of a rule should reflect the change in overall system RPP effected by that rule, the fitness update for a rule may not occur until after the rule's action has percolated through the system. The simplest way of achieving this is to update the fitness of the selected rule during the immediately following LCT lookup cycle, e.g. the rule whose action is performed during lookup cycle one would have its fitness updated during lookup cycle two. This gives every action a period of exactly T_{Act} to affect the system-wide RPP metric before the fitness of the action's proposing rule is updated.

Although a short action delay is enough for actions with fairly small response times (a few clock cycles, such as lowering a frequency to reduce the number of timing errors within a component), other actions (such as migrating a task from one processing core to another) may require a longer interval (hundreds to thousands of cycles) before the system returns to a stable state. If the fitness of all rules were updated one lookup cycle after their execution, T_{Act} would have to be chosen to accommodate the slowest action response. To allow for actions with very large response times (e.g. actions targeted at reducing the die temperature, which may take several seconds or millions of cycles to change), this results in an excessive action delay and a correspondingly low lookup frequency. Possible solutions to this problem include dynamically adjusting T_{Act} based on the action that was executed, or holding off the fitness update for several lookup cycles while other actions may be performed. Unfortunately, this last approach would result in a multi-step problem, for which a direct mapping between action and reward becomes difficult.

5 Conclusion and Future Work

In this paper we have presented an efficient hardware implementation of an LCS derivative called Learning Classifier Table (LCT), which is based on a new rule selection algorithm that combines match set creation and action selection into a single step. Initial synthesis results have revealed an affordable area overhead of less than five percent of a Leon2 CPU core in standard FPGA technology. Further extensions to the LCT were proposed in order to give the effects of actions with a large response delay enough time to percolate through the system. This prevents a reward from being given prematurely, and avoids unwanted action repetition when an action does not have an immediate impact on the monitor signals.

Future work involves the investigation of overlapping actions and their effect on system stability. Action overlap can occur not only because of simultaneously active actions initiated by a single LCT, but also from actions initiated by LCTs distributed across the system's other functional elements. This interplay of actions and its influence on system stability, as well as the steps necessary to guarantee a stable system, are the primary focus of our ongoing research.

Furthermore, an FPGA prototype of a functional system benefiting from learning classifier tables is essential as a demonstration platform of hardware capable reinforcement learning. We therefore plan to implement a multi-processor SoC based on Gaisler's Leon processor core [11], with additional bus, memory and I/O controllers, each being optimized and protected by an LCT.

Finally, the dynamic update of classifier rules via an interface to a software learning algorithm must be provided. This will allow for a full-fledged reinforcement learning system operating at hardware runtime.

Acknowledgements

This work is funded by the German Research Foundation (DFG) under the Organic Computing SPP-OC 1183 research program. We explicitly thank our partners W. Rosenstiel from University of Tübingen, O. Bringmann and A. Bernauer from FZI Informatik in Karlsruhe for their fruitful cooperation in our joint project on Autonomic Systems-on-Chip.

References

[1] A. Bernauer et al., "An Architecture for Runtime Evaluation of SoC Reliability", Informatik für Menschen, volume P-93 - Lecture Notes in Informatics, 2006, pp. 177-185
[2] A. Bouajila et al, "Organic Computing at the System on Chip Level", VLSI-SoC 2006, pp. 338-341
[3] J. Holland, "Adaptation", Progress in Theoretical Biology, vol. 4, 1976, pp. 263-293
[4] S. Wilson, "Classifier fitness based on accuracy", Evolutionary Computation, 3, 1995, pp. 149-175
[5] S. Wilson, "A zeroth level classifier system", Evolutionary Computation, 2, 1994, pp. 1-18
[6] M. Butz, "Documentation of XCSFJava 1.1 plus Visualization", MEDAL Report No. 2007008
[7] C. Bolchini et al, "Evolving classifiers on field programmable gate arrays: Migrating XCS to FPGAs", Journal of Systems Architecture 52, pp. 516-533
[8] P. S. Efraimidis, P. G. Spirakis, "Weighted random sampling with a reservoir", Inf. Process. Lett. 97, 5 (Mar. 2006), pp. 181-185
[9] Gregable, "Reservoir Sampling", http://gregable.com/2007/10/reservoir-sampling.html
[10] LFSR, e.g. "Wikipedia article on Linear feedback shift register", http://en.wikipedia.org/wiki/Linear_feedback_shift_register
[11] http://www.gaisler.com GR-CPCI-XC4V LEON Compact-PCI Development board