Mining Interesting Patterns from Hardware-Software Codesign Data with the Learning Classifier System XCS

Fabrizio Ferrandi, Pier Luca Lanzi, Donatella Sciuto
Dipartimento di Elettronica e Informazione
Politecnico di Milano
{ferrandi,lanzi,sciuto}@elet.polimi.it

Abstract- Embedded Systems are composed of both dedicated elements (hardware components) and programmable units (software components), which have to interact with each other for accomplishing a specific task. One of the aims of Hardware-Software Codesign is the choice of a partitioning between elements that will be implemented in hardware and elements that will be implemented in software is one of the important step in design. In this paper, we present an application of the learning classifier system XCS to the analysis of data derived from Hardware-Software Codesign applications. The goal of the analysis is the discovering or explicitation of existing interrelationships among system components, which can be used to support the human design of embedded systems. The proposed approach is validated on a specific task involving a Digital Sound Spatializer.

1 Introduction

The current trend in Embedded Systems (ES) design is moving toward the integration of increasingly complex applications on a single chip. An Embedded System, generally speaking, is composed of both dedicated elements (hardware components) and programmable units (software components), which have to interact with each other for accomplishing a specific task. Hardware-Software Codesign aims at proposing some methodology for a cooperative design of such mixed systems. According to the major companies involved in production of embedded systems, the ideal Codesign methodology shall support the exploration of the highest possible number of alternatives (in terms of hardware-software architectures) since the early design stages as this will prevent costly correction efforts in the deployment phase.

The choice of an architecture, i.e. of a collection of components that can be either programmable, re-configurable or customized, is one of the important steps in design. One of the major issue in Embedded Systems design concerns the search for the hardware-software configuration (i.e., partitioning) that better fits a set of performance constraints and cost limits. Different approaches have been proposed in the literature to support the design of embedded systems, but each of them is characterized by the same overall schema. Given an embedded system, first a formal model is developed to identify the various system functionalities and to describe how these functionalities interact. This formal model describes the space of the possible hardware-software partitioning and the corresponding costs (i.e. a cost function is defined). Finally, a search algorithm (e.g., a heuristic [3], a genetic algorithm [1, 10]) is applied to find the best partitioning which corresponds to the minimum value of the given cost function.

In this paper, we take a different viewpoint and apply learning classifier systems to extract interesting patterns from the cost function defined over space of the feasible partitioning. Our aim is not to find the best partitioning, instead we are interested in discovering rules of thumb which can be used both (i) to support a user in the partitioning process, and (ii) to discover interesting interrelationships among different model components. More specifically, we adopt the approach we introduced in [4] according to which: (i) the embedded system functionalities, and their interactions, are described at the highest possible level of abstraction, coherently with the system level view of the Codesign problem; (ii) the cost function is the result of a dynamic analysis of the system, performed via simulation, that considers both local and global features of the system under study. The resulting cost function maps a binary tuple, representing an hardware-software partitioning, to a real number, representing the cost of the partitioning. In particular, we consider the simplest cost function possible, i.e., the execution time. Instead of using this function as the target for an optimization algorithm (as usually done [3, 1, 10]), in this paper, we view this cost function from a Data Mining perspective and apply Learning Classifier Systems to extract interesting design patterns that underlie the definition of the function. The result of the analysis is a set of rules, the classifiers, which maps classes of hardware-software partitioning and design decisions into a modification of the overall cost. Separately, we also present some preliminary result involving the use of Neural Networks [11] for approximating the cost function from a very limited set of partitioning so as to decrease the amount of resources required both for our analysis and for the search of the best partitioning in [4].

The paper is organized as follows. In Section 2 we overview the hardware-software codesign approach we followed to generate the data for our work. In Section 3 we describe a specific task involving a Digital Sound Spatializer. In Section 4 we introduce learning classifier systems and specifically Wilson’s XCS which we use for the analysis we present here. In Section 5 we apply XCS to the data obtained through the Hardware-Software Codesign approach previously introduced and discuss the type of results that our analysis can produce. In Section 6 we present the initial results regarding the use of neural networks the analysis we performed on the data from hardware-software codesign. Finally, in Section 7 we outline some of what we believe are the most interesting future research directions.
2 Hardware-Software Codesign

We now shortly overview the Codesign approach, we introduced in [4], that we use to generate the data used for the analysis in this paper. We refer the interested reader to [4] for a detailed description of the overall Codesign approach.

In [4], the Hardware-Software Codesign process is organized into three main steps that lead to a final partitioning optimized with respect to the given metrics. First, the software performance of the different functional components is estimated from a first description of the application which, nowadays, is usually provided in terms of C-like programming languages. Then, the global and local system performance are estimated in terms of hardware description, communication interfaces, and mixed hardware-software architecture. In the final step, the above information is used to guide the algorithm which will provide the optimal partition.

Software Performance Estimation. This first step aims at giving both a characterization of the timing behavior of the system when specified in some programming language (nowadays C and C++ are the most widely used) and a mean to dynamically determine, via simulation, the best granularity at which the partitioning process is to be faced. The choice of the best granularity, is of paramount importance, as it defines how to decompose the system into sub-parts, which will then be the units over which the final partitioning phase will reason for defining the final system architecture. According to the approach we introduced in [4], see also [9], this problem is tackled by starting at the process level of granularity, which is a sort of meeting in the middle, between the whole system and the operation level. To accomplish this, the target micro-processor is selected and the executable code of the application is produced. Then a parsing tool identifies in the code, by creating a Data Flow Graph (DFG), all the data dependencies to be used for classifying the code fragments, based on which to dynamically identify the mathematical relations that describe execution time as a function of the input data. After having found these mathematical relations, for all the processes involved in the system, the designer, according to the timing system constraints, is able to understand if some of these processes are time critical for the application. For such processes, a finer grain granularity is obtained, by decomposing them in smaller sub-parts, which will be computationally manageable. The outputs of this phase are (i) the identification of the granularity for analyzing the system, based on which all the subsequent phases will be carried on, and, (ii) a characterization in terms of execution time of all the system sub-parts seen as software modules. This characterization is employed in the final simulation of the system seen as a mixed hardware-software architecture.

Hardware and Communication Performance Estimation. Given the software performance characterization of the system, and the selected granularity, in this second step we estimate both local and global system performance taking into account the possibility of realizing the different components both in hardware and in software. For gaining a fast estimation on hardware timing, the approach we introduced in [4] relies on the information collected by the parsing tool at the beginning of the software estimation phase. By considering the Data Flow Graph (DFG) of each system component identified by the dynamically determined granularity, we apply to each of them an unoptimized version of an unconstrained scheduling algorithm [8], which gives a characterization of hardware performance. The same approach is applied to model the communication among different classes of components (see [4] for details).

The Simulator. After having characterized all the components and all the possible communication channels a simulator is built. This is used to simulate several instances of the embedded system architecture, which differ one from the other on the hardware or software characterization of the single sub-parts. Each instance also carries the correct model of communication according to its specific mix of hardware and software parts. The simulator is written in SystemC and it is the exact translation of the formal model used to describe the target embedded system. It exploits both the local characterizations given by the previous software and hardware estimation phases and the communication model to compute global system performance when instantiating different possible combinations of hardware and software modules. At a very high level of abstraction, the simulator takes as input a binary string representing a partitioning and returns a cost. This mapping is used in [4] to support the search for the final partitioning. In contrast, in the work presented here, we analyze the input-output mapping obtained from the simulator to identify interesting patterns which characterize the embedded system.

3 Case Study

The application chosen as a case study is a Digital Sound Spatializer, which encompasses all the characteristics of a typical embedded system. A sound spatializer is a machine, be it analog, digital or hybrid, which takes as its input a sound which is acquired by one or more sound sources: e.g., from a musical instrument or a singer, which is usually recorded with high-quality noise-reduction microphones. Its output is the input sound as it would be perceived by someone in a room, in which the sound is supposed to be played. Such a device is used for musical applications and is often distributed with the most common commercial tools for digital audio processing. This device, then simulates the room, or its effect on audio perception by reverberating, echoing and delaying the input sound in a complex way. The target microprocessor we choose for testing the software behavior is the Xilinx MicroBlaze, which is a 32 bit RISC soft-processor core. For simulating the acquisition phase of the audio stream and the output of the reverberated sound we used an audio file and sampled it at 44.1 KHz extracting a 16 bits sample; once the sample has been processed by the application we wrote the resulting sample on
an output file. The code describing the application is composed of 14 processes, which describe all the different components of the Digital Audio Spatializer: the Multitap Delay line, which simulates both the direct sound and the discrete echoes, the Multitap Delay mixer which scales the output of the Multitap Delay line, the Sound Source, which is one of the processes modeling the room where the sound is diffused, the Walls (which for the final partitioning are considered as 4 different processes), the Listener, who receives the audio sample modified by the four walls, reassembles and scales it and the Output Mixer process, which receives the processed samples from the delay line and from the listener and reassembles it applying to it the final distortion according to the given parameters. After having completed the local characterization, and the simulator has been built, we run a complete simulation of all the possible system instances, which are, according to the 14 different processes we consider, 2^{14} = 16384. This simulation took 3 days and half to complete on an Intel, PIV, 1.7 GHz, 1 GB RAM. An extract of the overall table representing the cost function is depicted in Table 1.

4 The XCS Classifier System

Learning classifier systems (LCSs) are rule-based systems which exploit reinforcement learning [12] and genetic algorithms [5] to extract interesting rules (the classifiers) from a set of examples [7]. In the analysis presented in this work, we employ Wilson’s XCS [13], probably one of the best performing models of LCS available nowadays.

Classifiers in XCS consist of a condition, an action, and four main parameters: (i) the prediction \( p \), which estimates the payoff that the system expects when the classifier is used; (ii) the prediction error \( \epsilon \), which estimates the error of the prediction \( p \); (iii) the fitness \( F \), which estimates the accuracy of the payoff prediction given by \( p \); and finally (iv) the numerosity \( num \), which indicates how many copies of classifiers with the same condition and the same action are present in the population. Note that in the population \([P]\) no duplicates classifiers exist, i.e., there can be only one classifier with a certain condition-action-pair.

**Performance Component.** At each time step, XCS builds a match set \([M]\) containing the classifiers in the population \([P]\) whose condition matches the current sensory inputs; if \([M]\) contains less than \( \theta_{num} \) actions, covering takes place and creates a new classifier that matches the current inputs and has a random action. For each possible action \( a_i \) in \([M]\), XCS computes the system prediction \( P(a_i) \) which estimates the payoff that the XCS expects if action \( a_i \) is performed. The system prediction is computed as the weighted average of the predictions of classifiers in \([M]\), \( cl \in [M] \), which advocate action \( a_i \), i.e., \( cl.a = a_i \):

\[
P(a_i) = \frac{\sum_{cl.a = a_i, cl \in [M]} cl.p \times cl.F}{\sum_{cl.a = a_i, cl \in [M]} cl.F} \tag{1}
\]

where, following the notation of [2], \( cl.a \) identifies the action of classifier \( cl \), \( cl.p \) identifies the classifier prediction, and \( cl.F \) identifies the classifier fitness. Then XCS selects an action to perform; the classifiers in \([M]\) which advocate the selected action form the current action set \([A]\). The selected action is performed in the environment, and a scalar reward \( R \) is returned to XCS together with a new input configuration.

**Reinforcement Component.** When the reward \( R \) is received, the parameters of classifiers in \([A]\) are updated in the following order [2]: prediction, prediction error, and finally fitness. Prediction \( p \) is updated with learning rate \( \beta \) \((0 \leq \beta \leq 1)\); \( p \leftarrow p + \beta(R - p) \). Similarly, the prediction error \( \epsilon \) is updated as: \( \epsilon \leftarrow \epsilon + \beta([R - p] - \epsilon) \)

**Fitness Update.** The update of classifier fitness consists of three steps. First, the raw accuracy \( \kappa \) of the classifiers in \([A]\) is computed as:

\[
\kappa = \begin{cases} 
1 & \text{if } \epsilon \leq \epsilon_0 \\
\alpha(\epsilon/\epsilon_0)^{-\nu} & \text{otherwise}
\end{cases} \tag{2}
\]

A classifier is considered to be accurate if its prediction error \( \epsilon \) is smaller than the threshold \( \epsilon_0 \); a classifier that is accurate has an raw accuracy \( \kappa \) equal to one. A classifier is considered to be inaccurate if its prediction error \( \epsilon \) is larger than \( \epsilon_0 \); the raw accuracy \( \kappa \) of an inaccurate classifier is computed as a potential descending slope given by \( \alpha(\epsilon/\epsilon_0)^{-\nu} \). The parameter \( \epsilon_0 \) \((\epsilon_0 > 0)\) is the threshold that determines to what extent prediction errors are accepted; \( \alpha \) \((0 < \alpha < 1)\) causes a strong distinction between accurate and inaccurate classifiers; \( \nu \) \((\nu > 0)\), together with \( \epsilon_0 \), determines the steepness of the slope used to calculate classifier accuracy. The raw accuracy \( \kappa \) is used to calculate the relative accuracy \( \kappa' \) as:

\[
\kappa' = \frac{(\kappa \times num)}{\sum_{cl \in [A]} (cl.k \times cl.num)} \tag{3}
\]

where \( cl.k \) is the raw accuracy of classifier \( cl \), as computed in equation 2; \( cl.num \) is the numerosity of classifier \( cl \). Finally the relative accuracy \( \kappa' \) is used to update the classifier fitness as: \( F \leftarrow F + \beta(\kappa' - F) \).

**Discovery Component.** On regular basis, roughly every \( \theta_{ga} \) steps, the genetic algorithm is applied to classifiers in \([A]\). It selects two classifiers with probability proportional to their fitnesses, copies them, and with probability \( \chi \) performs crossover on the copies; then, with probability \( \mu \) it mutates each allele. The resulting offspring are inserted into the population and two classifiers are deleted to keep the population size constant.

5 Experimental Results

We apply the XCS classifier systems to mine interesting patterns from the cost function obtained by applying the Code-sign approach we described in Section 2 (see also [4]) to the Digital Sound Spatializer we illustrated in Section 3.

Each experiment consists of a number of problems that XCS must solve. For each problem, an input partitioning, represented by a binary string of length 14, is presented to XCS.
Based on the current input partitioning, XCS suggests an action which identifies a possible modification to the current partitioning through the flip of a bit in the current hardware-software configuration. The action is performed, and the current partitioning is modified. As a result, XCS receives a reward computed as the difference between (i) the cost of the current partitioning and (ii) the cost of the new partitioning obtained through the suggested modification. Thus XCS will receive a positive reward if the suggested action corresponds to a decrease in the cost of the partitioning, a negative reward if the suggested action corresponds to an increase in the cost of the partitioning. Accordingly, by maximizing the reward, XCS tends to develop classifiers which suggest improvements in the cost function.

Classifier conditions in XCS are represented as strings of 14 symbols (one for each component) over the ternary alphabet \( \{0,1,\#\} \); the symbol \( \# \), called don’t care, means that the corresponding position in the classifier condition can either be 0 or 1, i.e., the corresponding component can be implemented either in hardware or software. There are 14 possible classifier actions, numbered from 0 to 13, that represent the change of the corresponding bit in the current input partitioning. The performance of XCS is measured as the error between the actual reward received as effect of the proposed action, and the reward that XCS estimated. According to this settings, XCS learns to predict how the modifications of the input partitioning will influence the target cost function. More precisely, since our cost function is simply the estimated execution time, XCS learns to predict how the modifications of the input partitioning will influence the overall execution time.

Figure 1a reports the error over XCS prediction for the data derived for the Digital Sound Spatializer; population size \( N \) is 5000 classifiers and \( c_0 = 10^{-4} \), while all the other parameters are set as usual (e.g., [13]). As the figure show, XCS learns to predict quite accurately how the modifications in the current partitioning will affect the cost, i.e., the execution time. Figure 1b reports the percentage of classifiers in the population. Initially, the population rapidly grows while XCS is starting to learn, as the learning proceeds the population size shrinks and the number of classifiers in the population decreases showing that XCS is converging toward a minimal set of classifiers that represent some accurate piece of knowledge extracted from the cost function.

Table 2 reports some classifiers evolved during one of the runs depicted in Figure 1. The first classifier indicates that if component \( c_2 \) and component \( c_3 \) are implemented in software, then changing component \( c_2 \) to hardware (action is 2) will cause a reduction \( p \) in the cost (i.e., in the execution time) equal to 0.61, and that this prediction is affected by an absolute error (\( \epsilon \)) of \( 1.384 \times 10^{-4} \). Note that the second and the third classifiers provide complementary information. The second classifier suggests that if component \( c_0 \) and \( c_1 \) are implemented in hardware, then changing component \( c_1 \) to software will cause an increase of 0.76 in the cost (we remind the reader that a negative prediction means that the resulting partitioning has an higher cost). Conversely, the third classifier suggests that if component \( c_0 \) is implemented in hardware and component \( c_1 \) is implemented in software, the changing component \( c_1 \) to hardware (so to obtain a partitioning matched by the first classifier), will cause a decrease of 0.76 in the cost. Note that the classifiers reported in Figure 2 represent high level and accurate information about the cost function; all the classifiers in table 2 are very general in that they apply to many partitioning (i.e., they have many don’t care symbols); in addition they are very accurate since their prediction error \( \epsilon \) is very small if compared to the prediction value. The evolved classifiers can be used to improve the designer understanding of the existing interactions among different system components, either as an effective support to the search of the best partitioning.

### 6 Using Neural Networks to Approximate the Cost Function

Reinforcement learning methods, such as learning classifier systems, assume that it is always possible to provide a reward signal to the agent, whatever the situation that the agent is experiencing, whatever the action that the agent performs. In our case, this requires that for any possible hardware-software configuration, and for any possible modification that the system suggests, it is always possible to provide the agent a reward value which estimates the modification in the overall configuration cost if the modification is performed. In our case study, this has been obtained by computing the cost of every possible input configuration through computer simulations and storing the resulting cost function into a lookup table with \( 2^{14} \) entries.

As the complexity of the embedded system increases, it easily becomes infeasible to compute the whole cost function since the number of configurations to compute, and simulations to run, is exponential in the number of components involved. To cope with the high dimensionality of the input space, we can use a function approximator, to derive an approximation of the whole cost function from a limited number of example. In this section, we present a preliminary set of, very promising, results regarding the use of artificial Neural Networks to approximate the cost function for the sound spatializer discussed in Section 3. For this purpose, we apply a standard model of multilayered Neural Network [11], with 14 input nodes (one for each components), one output node with a linear activation function, and respectively five or ten hidden nodes with hyper-tangent activation functions. Each neural network is trained respectively with the 1%, 5%, 10%, and 20% of the overall cost function, and tested on the whole function.

Figure 2 reports the train and test errors of a neural network with five hidden nodes when this is trained respectively on the 1%, the 5%, the 10%, and the 20% of the whole cost function; Figure 2a reports the error on the training set; Figure 2b reports the error on the test set; curves are averages over ten experiments. Figure 3 reports the train and test errors of a neural network with five hidden nodes when this is trained respectively on the 1%, the 5%, the 10%, and the
20% of the whole cost function; Figure 3a reports the error on the training set; Figure 3b reports the error on the test set; curves are averages over ten experiments. The figures show that all the models of neural networks reach, more or less, the same values or error on the whole function. When only the 1% of the overall function is used for training, the convergence is slower; but the final error value is quite near to that achieved with larger training sets. Note also that the performance of the net with ten hidden nodes (Figure 3b) is slightly better than that with only five nodes (Figure 2b).

Overall these (preliminary) results suggest that neural networks might be very effective for developing accurate approximation of the cost function from a very limited number of examples (as few as the 1% of the overall function) so as to support the search of the optimal partitioning (see [4]) and the mining of patterns, as done in this paper. However, the approach needs to be validated on more case studies since, although the Digital Sound Spatializer encompasses all the characteristics of a typical embedded system, its complexity in terms of components is quite limited.

### 7 Future Research Directions

We presented preliminary results regarding the use of learning classifier systems, and more specifically of Wilson’s XCS, to mine interesting knowledge from Hardware-Software Codesign data. The results show that XCS extracts rules (classifiers) that represent high level and accurate knowledge about the cost function which can be used either to guide the search for the best partitioning, either to highlight to a human designer interesting interrelationships among hardware-software components. We noted that learning classifier systems (as well as all the other reinforcement learning techniques) in principle require the knowledge of the whole cost function. But this is infeasible in practice since it would require a number of simulations that is exponential in the number of components involved. Accordingly, we show that neural networks can be effective in developing an accurate approximation of the whole cost function from a limited number of simulations. Although the results we presented here are promising, they need to be validated on more Codesign applications. Even if we are quite confident that XCS is quite effective in mining rules from data (e.g., [6, 14]), in this particular case, its performance on large problems will rely heavily on the possibility of having accurate approximations of the cost function. This, according to the approach we presented here, will depend on the effectiveness of neural networks in developing accurate approximations of cost functions for complex embedded systems. But this needs further experiments.

### Bibliography


Table 1: An extract from the cost function computed for the Digital Sound Spatializer: $c_j$ represents the $j^{th}$ component; $c_j$ is 0 if the component is implemented in hardware, 1 if the component if implemented in software; Cost is the overall execution time estimated through the simulation (see Section 2). Execution time is shorter when all the components are implemented in hardware, longer when all the components are implemented in software.

<table>
<thead>
<tr>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
<th>$c_6$</th>
<th>$c_7$</th>
<th>$c_8$</th>
<th>$c_9$</th>
<th>$c_{10}$</th>
<th>$c_{11}$</th>
<th>$c_{12}$</th>
<th>$c_{13}$</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2.03</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2.08</td>
</tr>
<tr>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2.08</td>
</tr>
<tr>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4.10</td>
</tr>
<tr>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5.00</td>
</tr>
</tbody>
</table>

Figure 1: XCS applied to the cost function for the Digital Sound Spatializer: (a) error on the prediction of the cost; (b) percentage of classifiers in the population. Curves are averages over 10 experiments.

Figure 2: Multilayered neural network with five hidden nodes trained on the 1%, the 5%, the 10%, and the 20% of the whole cost function: (a) error on the train set; (b) error on the whole function. Curves are averages over 10 experiments.
Figure 3: Multilayered neural network with ten hidden nodes trained on the 1%, the 5%, the 10%, and the 20% of the whole cost function: (a) error on the train set; (b) error on the whole function. Curves are averages over 10 experiments.


