XCS Applied to Mapping FPGA Architectures

Martin Danek
Dept. of Computer Science and Engineering
Czech Technical University in Prague
Faculty of Electrical Engineering
Czech Republic
Email: danek@sun.felk.cvut.cz

R. E. Smith
Intelligent Computer Systems Centre
University of The West of England, Bristol
Computing, Engineering and
Mathematical Sciences Faculty
United Kingdom
Email: robert.smith@uwe.ac.uk

Abstract

This paper considers the application of XCS to the complex, real-world problem of mapping Boolean networks to technology-specific layout of field programmable gate arrays (FPGAs). The mapping is formulated as a temporal task, where the XCS's actions are to create blocks (based on an abstract Boolean network) that can be placed in the FPGA, one-at-a-time. Despite the complexity of this task, we demonstrate that the system is effective. We demonstrate the transfer of knowledge stored in an XCS rule set from a small FPGA mapping problem, to a larger problem. We present a novel technique that utilizes a problem-specific finite state machine and two populations of classifiers to treat the problem hierarchically. Final sections of the paper discuss implications and future directions for this work.

1 INTRODUCTION

Field-programmable gate arrays (FPGAs) are semi-custom VLSI circuits that were first introduced by the Xilinx company in 1984. They usually consist of a two-dimensional matrix of configurable logic blocks (CLBs) surrounded by special input-output blocks (IOBs) on the perimeter of the CLB matrix. An important part of the chip is formed by a programmable interconnect that can be used to connect inputs and outputs of logic blocks to form a desired circuit. The granularity of the logic blocks differs from fine-grain (universal two-input logic gates in the Xilinx XC6200 family) to coarse-grain (two four-input and one three-input cascaded look-up tables (LUTs) in the Xilinx XC4000 family).

FPGAs are favored for their short design cycle (see Figure 1), since one design cycle (one design implementation) takes hours rather than days (in the case of mask-programmable gate arrays), or months (in the case of application-specific integrated circuits, ASICs). This together with their low cost makes them suitable both for low-volume production and for prototyping of ASICs.

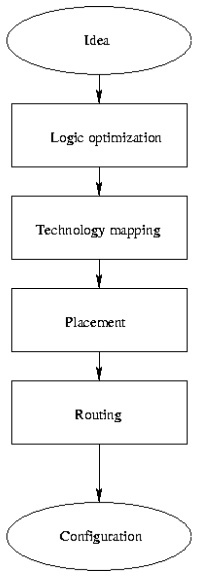

Figure 1: A typical FPGA design cycle.

Current field programmable gate arrays are very complex devices. Design software is needed to assist the user in implementing a circuit. This software usually consists of several subsequent optimization routines that transform the information about the circuit to be implemented (the *netlist*) to a more device-specific form.

These routines are usually referred to as

- the *mapper*,
- the *placer* and
- the *router*.

The software-assisted design procedure is typically as follows:

1. A design is specified by an abstract Boolean network (ABN) of logic gates and flip-flops.
2. The mapper performs a device-independent logic optimization. This transforms logic functions so as to obtain their minimal form.

3. Then the mapper maps the design to the target technology, which means that the gates in the netlist that represent the minimized logic functions and flip-flops are translated to blocks that are supported by the target FPGA (up to five-input LUTs and D-type flip-flops (DFFs) in the case of XC4000 devices).

4. The mapper forms groups of these blocks that fit into one CLB. Each CLB in the XC4000 devices can implement two four-input LUTs (usually referred to as F-LUT and G-LUT), one three-input LUT, (usually referred to as H-LUT), and two DFFs, (usually referred to as DXFF and DYFF). See Figure 2.

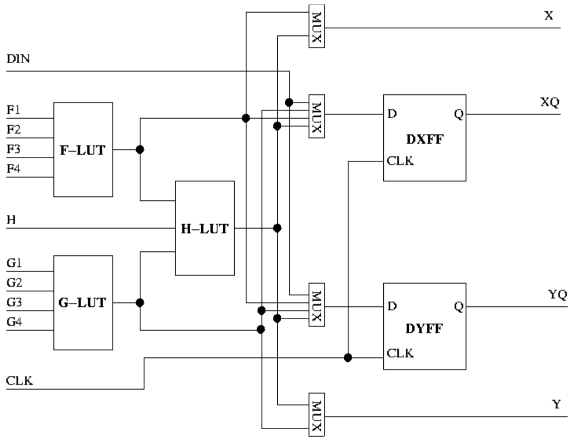

Figure 2: An internal structure of a configurable logic block (XC4000).

5. The placer assigns these groups (in fact, CLBs) physical positions in the CLB matrix of the target FPGA.

6. The router determines the way the CLBs are connected by universal wire segments that implement signal networks from the netlist. This consists of assigning wire segments to individual signal networks and of connecting these segments together.

7. As a final step, based on all this information, a device configuration bit stream is generated that can be downloaded to the FPGA, such that it is programmed to the desired function.

2 THE MAPPING TASK

The mapping phase takes an interconnection of abstract operators (the subject Boolean network) and generates an interconnection of logic cells selected from a given library. For instance, in the case of the Xilinx XC4000 devices, these are up to four-input look-up tables and edge-triggered D-type flip-flops.

The task of technology mapping in the case of LUT-based FPGAs is to find a set of clusters of technology dependent units that cover the whole Boolean network. A good cluster selection mechanism should take into account both the internal structure of the configurable logic block (CLB) of the target device (for an example, see [15]) and routing delays, in order to achieve good area and performance results.

The mapping task is an NP-complete problem [8]. The prevailing way to tackle it is to implement a heuristic algorithm, based on a previously gained human knowledge of the target architecture. This approach may be limiting for more complex devices, because they may contain features not recognizable at first sight. In addition, there is the question of considering signal delays in the mapping phase. Up-to-date mappers approximate delays with a unit delay per one level of logic, or they use signal delay values that were generated in previous iterations [4]. The unit-delay approach is very fast and is optimal in situations where the design topology is a sort of a planar graph. The iterative approach is time-consuming, because it requires several mapping runs interleaved with running placement and routing algorithms, and it does not guarantee the exactness of delay estimations, because they are reliable only in cases when small modifications take place.

The set of targeted logic cells in the mapping phase can be given either by an enumeration (i.e., as a set of standard cells (functions) for ASICs), or by a prescription (i.e., by specifying the maximal number of inputs a function may have, given that the function itself can be anything).

The first case is called *library-based mapping*, and it is suitable for all technologies that have fixed and relatively small libraries. For FPGAs, and especially for FPGAs based on look-up tables, this approach is not feasible, as the number of functions (i.e., library elements) grows rapidly with the increasing number of LUT inputs.

Currently, *prescription-based mapping* is very popular for LUT-based FPGAs. The drawback is that while known methods permit only simple prescriptions (commonly a single LUT); the logic blocks of existing FPGAs can be configured for more complex structures that are comprised of multiple LUTs. FPGAs with this ability (for example the Xilinx XC4000) are called *heterogeneous.*

The motivation for using heterogeneous FPGAs is a compromise between predictability of future routing (by increasing the number of routes with predictable delays - the ones hidden inside configurable logic blocks) and wasting logic that cannot be used (due to a fixed internal structure of logic blocks).

A mapping algorithm for heterogeneous FPGAs should combine the properties of both the approaches outlined above. Fortunately, the number of distinct configurable structures (regardless of the actual LUT contents) is conveniently small. The description can then be characterized as an enumeration of prescriptions. Although the algorithm that generates all clusters to cover the abstract Boolean network (see Figure 3) is relatively straightforward, the number of generated clusters increases with the complexity of physical cells and the proper cluster selection constraints become more complicated [10].

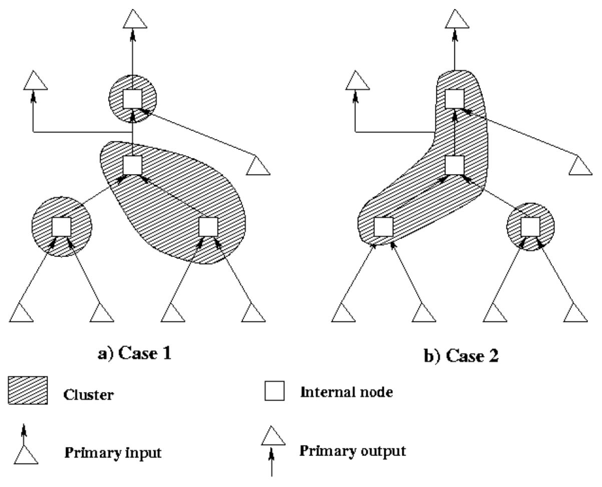

Figure 3: Covering an ABN with clusters - two cases.

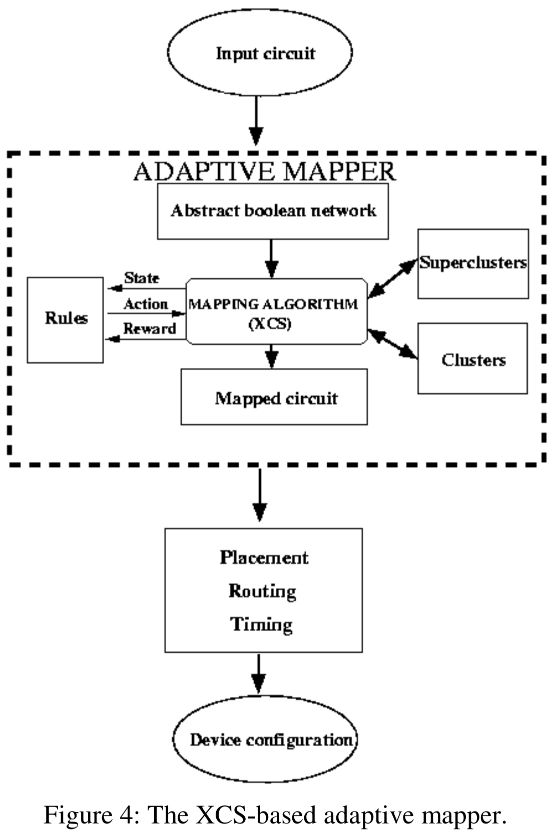

Figure 4: The XCS-based adaptive mapper.

3 ADAPTIVE MAPPING WITH XCS

This paper presents a rule-based, single-pass, adaptive mapper (see Figure 4) based on Wilson's XCS classifier system [12]. The XCS system was chosen due to its relative simplicity and ease of analysis.

It is intended that the mapper will operate by first being trained on benchmark circuits, such that it evolves a set of general mapping rules that perform well for any design, then using these pre-evolved rules to map a different, user-specified design in a single pass.

In the implementation presented here, the classifier system calls elementary mapping actions, and it attempts to evolve sequences of rules that lead to minimal CLB usage and small critical signal path delays. The approach used is similar to the FSM worlds used by Barry [2], but here the situation is more complicated. The underlying graph is not linear, but it is a directed tree with much longer start-to-terminal node distances, and some mapping actions group several nodes together and thus directly modify the environment. If all goes well, the adaptive nature of the classifier system should ensure that the final rules take into account global properties of the FPGA chip. Initially, the mapper is aware only of local properties of the chip - the internal architecture of CLBs. At the end of the run, we intend to be able to assemble efficient sequences of actions for the mapping task.

4 IMPLEMENTATION DETAILS

The structure of the XCS adaptive mapper is outlined in the following sections.

4.1 CONDITIONS

At any given stage of the mapping process, the state of the problem is described to the XCS by a *sense vector,* against which classifier conditions are mapped.

The sense vector used here consists of equal groups of binary flags, with each group representing the current CLB to be processed (the CLB that the XCS will place next), and CLBs connected to the inputs and outputs of that CLB. At present, only the maximum of four input and four output blocks are considered. Each group contains the flags representing the following binary characteristics of the associated block:

- addable to the current F-LUT, G-LUT, H-LUT,
- addable to the current DXFF, DYFF,
- connected to the current F-LUT, G-LUT, H-LUT (‘current F-LUT’ means F-LUT of the CLB that will be generated by the next GenCLB action, etc.)
- connected to the current DXFF, DYFF.

The sense vector also contains an extra group of flags that validate the previous groups, and that reflect the utilization of LUTs and DFFs of the CLB currently under construction (see Figure 5).

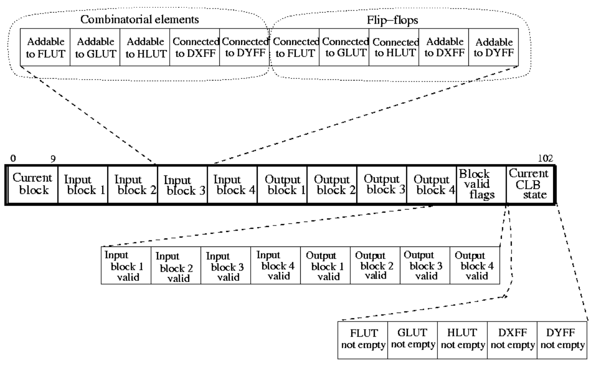

Figure 5: Diagram of sense vector that represents the problems state to the classifier system.

The length of the resulting sense vector is 103 bits. Although this represents a large state space, the advantage of such a coding is the clear meaning of genetic operations performed by the classifier system.

4.2 ACTIONS

The XCS adaptive mapper has three groups of mapping actions:

- actions that construct clusters (that stand for look-up tables (LUTs) or flip-flops) and assign them to the current supercluster (a supercluster is equivalent to a CLB),
- actions that generate the current supercluster, and
- actions that modify the current position in the netlist and an auxiliary *do nothing* action.

The first group contains actions:

- expand F-LUT, G-LUT, H-LUT, which assign a block to an empty cluster (LUT) or add a block to an existing cluster,
- assign a flip-flop to DXFF, or DYFF.

The only action in the second group takes clusters that were assigned to the current supercluster, generates the supercluster and empties all clusters:

- generate a CLB.

The third group consists of eight actions that implement different *move backward* and *move forward* commands, and one *do nothing* action.

The total number of actions available to the mapper is 15. The XCS software uses only the mutation operator for actions; and it randomly generates a new action number from the 15 alternatives.

4.3 REWARDS

The processing of rewards is based on reinforcement learning techniques [1]. To formulate the reward function correctly, it is necessary to declare the goals of the task correctly. The final goal is to generate efficiently a high-performance mapping of a design, which means:

- to use CLBs to their capacity
- to use as few CLBs as possible,
- to minimize the critical path delay, and
- to accomplish it in as few steps as possible.

This suggests that the reward should reflect the decrease in the number of CLBs used in the design and the decrease in the critical path delay. Each (unsuccessful) execution of an action should be given a negative reward.

A closer analysis of the interaction of the actions suggests the introduction of another goal: using CLBs to their capacity. Therefore, the reward received on a successful completion of an action was formulated as a weighted combination of factors:

$$\begin{aligned} reward &= \left(w_{num} * \Delta numCLBs\right) \\ &+ \left(w_{delay} * \sum\left(\Delta path\ delays\right)\right) \\ &+ \left(w_{usage} * CLBusage\right) \end{aligned}$$

where $\Delta numCLBS$ is the change in the number of CLBs used by the design at a time step *t*, each $\Delta pathdelay$ is the change in the delay of a single (two-point) signal path delays at a time step *t* (and the sum is taken over all such signal paths), and *CLBusage* is the utilization of the possibly generated CLB (or zero) (see Figure 6). To evaluate the reward function the number of CLBs (each unassigned block is declared to occupy one CLB) and actual values of path delays are calculated after an execution of each action. So far only a simple unit-delay model (see Section 2) is used, it is planned to replace it with a more realistic estimation of individual signal network delays in the future.

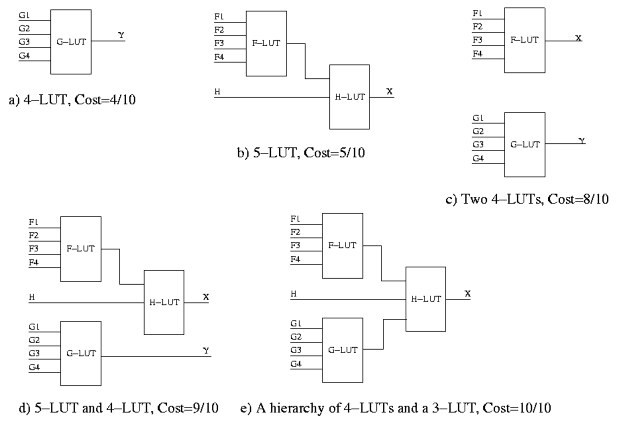

Figure 6: Different CLB configurations and their cost.

5 EXPERIMENTS

The target architecture for all runs presented here is the XC4000 CLB, the delay estimation used the unit delay model, and the evaluation was done for the LGSynth91 (also known as MCNC91) benchmark circuits [16]. The benchmarks come in the widely accepted BLIF format and are translated to the Xilinx XNF format to be compatible with the Xilinx tools.

Parameter values used in experiments are shown in Table 1:

Parameter	Value
w_{num}	7
w_{delay}	12
w_{usage}	4
β (learning rate)	0.2
γ (discount factor)	0.8
Exploration policy	Constant
Exploration probability	0.2

Table 1: Parameters for experiments presented here.

The constant exploration strategy [14] was used, population size was 8000. Each explore run (a complete mapping) was followed by one exploit run of the mapper. One experiment consisted of at most 80000 mapping trials. Each mapping trial was terminated when either all blocks were assigned to CLBs, or when 200 actions were executed. All plots shown contain values only for the exploit runs.

Benchmark	#gates	#DFFs	#ABN Nodes
MUX	61	0	61
MODULO12	16	4	20
DK16	135	5	140

Table 2: Benchmarks and their parameters

5.1 ADAPTING TO A SPECIFIC ARCHITECTURE

The principal results of this experiment are shown in Figure 7 and Figure 8 (two figures are shown to demonstrate the performance for examples of combinatorial and sequential circuits). Note that each of these results is from a single run. The performance plotted in these graphs has a direct relation to the performance of the resulting circuit and thus of the mapper, it shows a cumulative reward (see Section 4.1) over one mapping trial. Note that the actual performance plots are a 50 point moving average (as is typical in XCS performance plots), but the best performance plots may be a more realistic performance measures on this task, since rule sets are intended to be extracted, and used offline. A statistical overview of the learning task is shown in Figure 9 and Figure 10, which are averages over 10 learning runs. Note that these results are not subjected to a moving average, since that would tend to cloud the meaning of error bars.

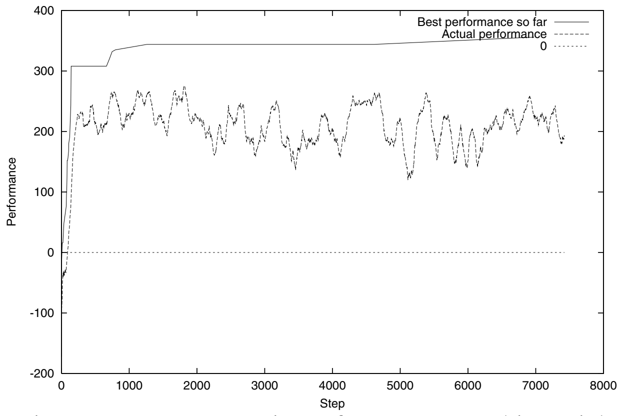

Figure 7: Improvement in performance - combinatorial circuit (MUX). Results from a single run, in training mode.

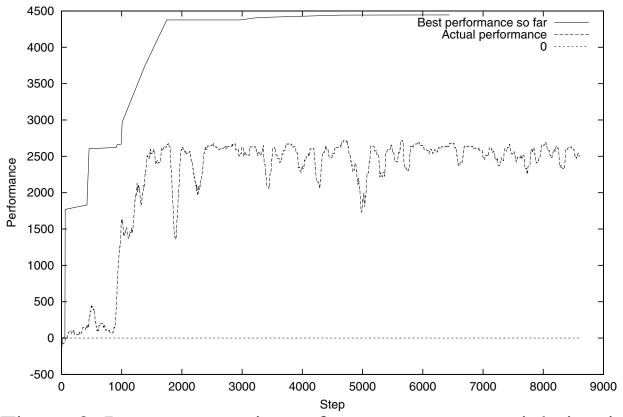

Figure 8: Improvement in performance - sequential circuit (DK16). Results from a single run, in training mode.

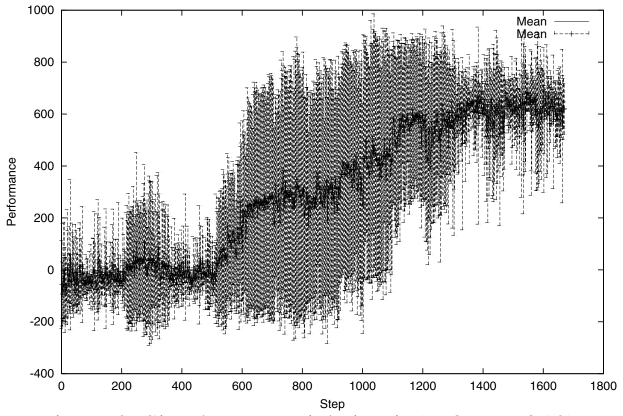

Figure 9: Simple sequential circuit (MODULO12), training mode, 10 independent runs, mean and standard deviation.

At the beginning, the mapper uses randomly generated mapping rules. This corresponds to the initial region with low performance. Then, as more mapping trials are performed, the quality of the mapped circuit, in terms of the performance, improves.

Note the differences between Figure 7 and Figure 8. The initial flat region lasts for more mapping steps in the case of the sequential circuit than for the combinatorial circuit.

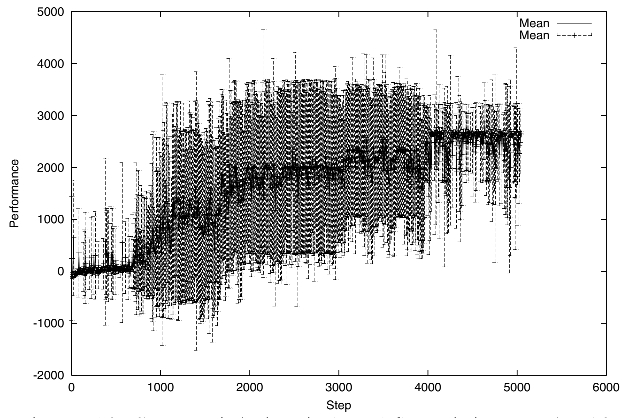

Figure 10: Sequential circuit (DK16), training mode, 10 independent runs, mean and standard deviation.

This is because learning a good mapping policy for sequential circuits is more difficult than for combinatorial circuits. Sequential circuits employ an additional logic element not found in combinatorial circuits (a flip-flop), and the critical path in such circuits is formed by stages of combinatorial circuits connected by flip-flops, which means that changes caused by mapping actions have a more local effect. The differences between Figure 9 and Figure 10 show that simple circuits can be mastered more quickly and efficiently than more complex circuits.

5.2 KNOWLEDGE TRANSFER

In this application, it is desirable that the evolved rule-sets can be shared among different designs, once they are discovered. The results in Figure 11 show that this is feasible for one design. Results in Figure 12 show that rules evolved for a simpler design can be used to improve the performance of the mapper for a more complicated design. Compare these results to the early generations of Figure 8, where no pre-evolved rules were included.

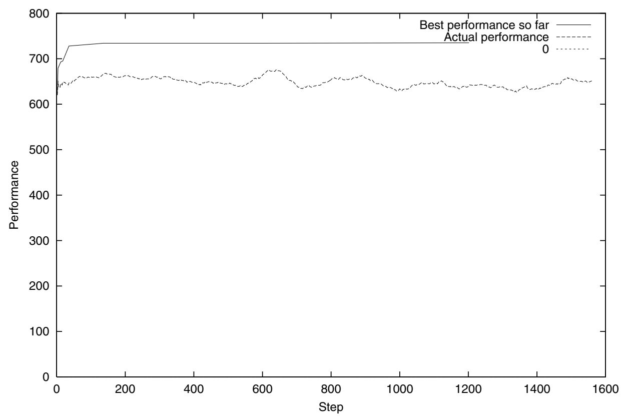

Figure 11: Results of using rules evolved in an experiment that started with an empty rule-set on the same simple sequential circuit. Pre-evolved rules for this experiment were taken from step 8000 of the training run.

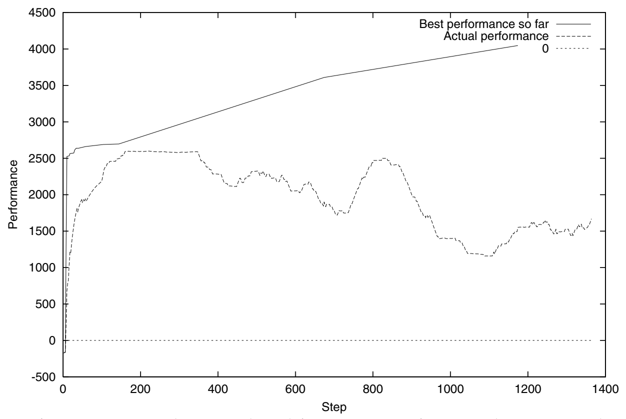

Figure 12: Rules evolved in an experiment that started with an empty rule-set on a smaller circuit – MODULO12 used on a larger sequential circuit (DK16). Pre-evolved rules for this experiment were taken from step 8000 of a training run.

6 DISCUSSION

Despite the relative success shown in these experiments, the task faced by the adaptive mapper is a complex one, and methods of reducing these complexities must be explored. Aspects of the task that may impede XCS include:

- The task is highly sequential, which means that the classifier system has to evolve long chains of rules.
- The mapping task environment is non-Markovian,
- The states in the search space are visited far from uniformly in the learning process, which may cause difficulties in Q-learning [1]. An improvement might be to consider only those states that are important to find a good policy; the problem is they are not known in advance. In general, there may be an exponential number of such states, because they are in tight connection with all possible mappings of the abstract Boolean network.
- The actions used by the classifier system have a different probability of triggering a reward. The reward is most often generated by the *generate CLB* action, but the effect of this action depends on all other actions in the (long) action chain that are not rewarded most of the time. The Q-learning mechanism is intended to ensure that the reward is distributed equally among all actions to reflect their effects, but the previously mentioned complications may severely hamper this effect.

These observations suggest that the actions may be better viewed as a hierarchy, according to the probability of receiving a reward from the environment:

- Lowest level actions: positioning actions - *moveBk, moveFw*,

- Mid-level actions: cluster generation - *assign DFF, expand LUT*,
- Highest level actions: supercluster generation - *generate CLB*

It may also be possible to improve the efficacy of the classifier system approach either by reducing the state space (omitting some information from the sense vector), or by 'hard-wiring' some a priori knowledge in the classifier system structure. The former approach is undesirable, since it would only increase the amount of hidden information in this non-Markov environment. The second approach appears more viable.

In response to these observations, a modification of the structure of the classifier system is introduced in the following section.

7 MULTI-POPULATION CLASSIFIER SYSTEM

The classifier system modifications introduced here are based on the action hierarchy discussed above, and on ideas presented in [11]. Also see [2] and [12].

In typical classifier systems there is only one population of rules (or a set of actions) and the system considers all rules (and the associated actions) at every time step. This approach presents difficulties in the mapping task, since the classifier system has to learn an efficient ordering of actions that can be deduced beforehand.

Simply stated, an optimal sequence of actions would be of the form

1. Repeat the following for some (unknown) number of steps
    a. move from the current position to a neighboring position
    b. decide whether the current position is to be a part of any of the currently generated clusters
2. generate a CLB.

Each high-quality sequence contains *move to a neighbor block* and *assign to/expand a cluster* actions, and is terminated with a *generate CLB* action. After performing a non-positioning action (i.e., cluster formation) at a position, it does not make sense to perform another non-positioning action, because a block can be part of only one cluster. On the other hand, it makes sense to perform a positioning action directly after another positioning action, for the same reason. Given this prerequisite knowledge, one can easily divide the actions the classifier system works with into several sets. One can supply a finite state machine (FSM) that uses the sense vector and a history of past actions to switch among the sets. This results in reducing the information that the classifier system has to discover, because it is contained in the FSM. It also reduces the search space that needs to be sampled by the genetic algorithm, because the action subsets are usually smaller than the original action set, and because the sense vector can be reduced according to the character of the actions in each set. The reward is calculated as if there was only one set of actions, so that the action sets influence each other directly.

To implement this idea in the adaptive mapper, we construct one population of positioning actions, and another population of all other actions (supplemented with a *do nothing* action). The FSM has only two states (each of which corresponds to one of the two populations). The FSM switches states when the last action was a non-positioning action, or the last action was a positioning action and the current block has yet to be included in any cluster.

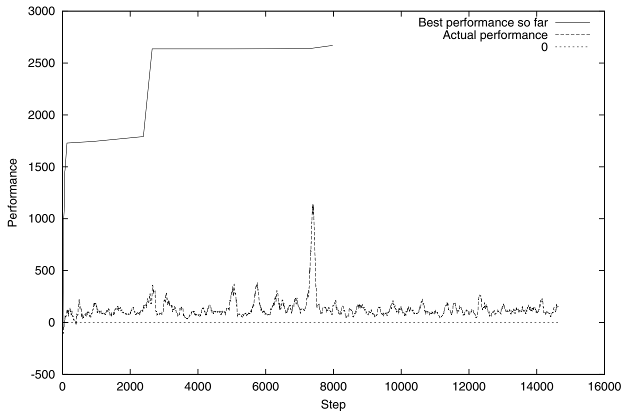

Figure 13: Single population mapper - sequential circuit (DK16), training mode, single run.

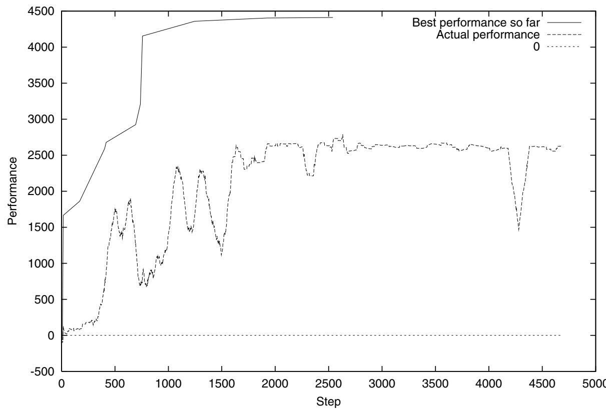

Figure 14: Two population mapper - sequential circuit (DK16), training mode, single run.

The performance of equivalent single- and multi-population mappers is shown in Figure 13 and Figure 14, respectively. Clearly, the two-population mapper performs better than the single-population mapper (note the ‘Actual performance’ curves rather than the ‘Best so far’ curves). The results shown are for single runs, to clarify behavior, but they are typical of many runs that we have performed.

8 FINAL COMMENTS AND FUTURE DIRECTIONS

Although this is a work in progress, at least two promising results have been described. First, it has been demonstrated that knowledge gained through XCS training on a small problem can improve performance of the system on a larger problem. Second, a novel approach to hierarchical tasks, using two populations and a problem-specific finite state machine, has been demonstrated to be effective. On the other hand, it is fair to say that any conventional heuristic would perform better than XCS at this moment, mainly because a heuristic was fully adapted to the problem domain by a human programmer. Clearly, given the complexity of this task, more study is needed. In addition to considering additional cases, several points need to be further addressed, including:

- Different methods of credit assignment, particularly epochal schemes, given the episodic nature of this task.
- Consideration of modifications to the classifier system, including the introduction of memory, and examination of ZCS [2], [7].

There are also key concepts in this work that deserve further broader consideration in other applications. These include knowledge transfer from simpler to more complex problems. Another promising area is the use of FSMs and multiple classifier populations to deal with hierarchical tasks. The application presented here used problem-specific information to construct the appropriate FSM, and its interactions with the classifier populations. Such problem-specific design is always good practice in GA-based applications. However, one can also imagine exploring this technique more generally, and allowing the FSM itself to be a subject of adaptation. This remains an interesting area for future investigation.

Acknowledgments

The authors would like to thank Larry Bull for useful and ongoing discussions that are helping to shape this research effort.

References

[1] Barto, A. G., Sutton, R. S. (1998). *Reinforcement learning*. MIT Press.

[2] Barry, A. (2001) A Hierarchical XCS for Long Path Environments. In L. Spector et al. (eds) *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference.* Morgan Kaufmann, pp 913-920

[3] Bull, L., Hurst, J. (in press). ZCS: Theory and practice. *Evolutionary Computation.*

[4] Cong, J., Ding, Y., Gao, T., Chen, K. (1994). LUT-based FPGA technology mapping under arbitrary net-delay models. *Computers and Graphics.* 18(4).

[5] Danek, M., Muzikar, Z. (1999). Global routing models. In Lysaght, P., Irvine, J., Hartenstein, R. (eds.) *Field-Programmable Logic and Applications: 9th International Workshop, FPL'99*. Springer Verlag.

[6] Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley.

[7] Lanzi, P. L. (1997). A model of the environment to avoid local learning with XCS in animat problems. *Technical report no. 97.46, Dip. Di Elettronica a Informazione, Politecnico di Milano.*

[8] Murgai, R. , Brayton, R. K., Sangiovanni-Vincentelli, A. (1995). *Logic synthesis for field-programmable gate arrays.* Kluwer.

[9] Servit, M., Muzikar, Z. (1994). Integrated layout synthesis for FPGAs. In Hartenstein, R., Servit, M. (eds.) *Field-Programmable Logic: Architectures, synthesis and applications: 4th International Workshop on Field Programmable Logic and Applications, FPL '94* .Springer Verlag.

[10] Servit, M., Yi, K. (1997). Technology mapping by binate covering. In Luk, W., Cheung, P. Y. K. (eds.), *Field-Programmable Logic and Applications: 7th International Workshop, FPL'97*. Springer Verlag.

[11] Wiering, M., Schmidhuber, J. (1997). HQ-learning. *Adaptive Behavior 6*: (2), MIT Press, pp 219-246.

[12] Wilson, S.W. (1987). Hierarchical Credit Allocation in a Classifier System. In Davis, L. (ed.) *Genetic Algorithms and Simulated Annealing*. pp. 104-115. Pitman.

[13] Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation.* 3(2).

[14] Wilson, S. W. (1996). Explore/exploit strategies in autonomy. In P. Maes, M. Mataric, J. Pollack, J.-A. Meyer and S. W. Wilson (eds.), *From Animals to Animats 4: Proceedings of the Fourth international Conference on Simulation of Adaptive Behaviour.* MIT Press.

[15] Xilinx (2001). Programmable logic data book (Xilinx)

[16] Collaborative Benchmarking Laboratory at North Carolina State University - Web Pages [online] http://www.cbl.ncsu.edu