Multi-objective Intrinsic Evolution of Embedded Systems

Paul Kaufmann and Marco Platzner

University of Paderborn, Department of Computer Science,
Warburger Str. 100, 33098 Paderborn, Germany
{paul.kaufmann, platzner}@upb.de

Summary. The evolvable hardware paradigm facilitates the construction of autonomous systems that can adapt to environmental changes, degrading effects in the computational resources, and varying system requirements. In this article, we first introduce evolvable hardware, then specify the models and algorithms used for designing and optimizing hardware functions, present our simulation toolbox, and finally show two application studies from the adaptive pattern matching and processor design domains.

1 Evolvable Hardware – An Introduction

In the last decades, *natural computing* methods which take problem solving principles from nature have gained popularity. Among others, natural computing includes evolutionary computing. Evolutionary computing covers population-based, stochastic search algorithms inspired by principles from evolution theory. An evolutionary algorithm tries to solve a problem by keeping a set (population) of candidate solutions (individuals) in parallel and improving the quality (fitness) of the individuals over a number of iterations (generations). To form a new generation, genetically-inspired operators such as crossover and mutation are applied to the individuals. A fitness-based selection process steers the population towards better candidates.

Evolvable hardware (EHW) denotes the combination of evolutionary algorithms with reconfigurable hardware technology and run-time optimization to construct self-adaptive and self-optimizing hardware functions (circuits). The term evolvable hardware was coined by de Garis [1] and Higuchi [2] in 1993. The essence of EHW is the usage of optimization algorithms during the run-time of a system to adapt its functionality to time-variant conditions. EHW is typically applied to two classes of hardware functions. The first class comprises functions with a performance that is dependent on the input data distribution, thus opening up the need for self-adaptation. Functions that should withstand partial defects of the computational resources or react to changes in the resources form the second class of EHW compliant functions. Common to both

classes is that the optimal solution is either unknown, too cumbersome to compute a-priory, or too complex to be formalized as a consolidated function.

The long-term goal of our work is the development of autonomous embedded systems that implement hardware functions characterized by their functional quality and resource demand [3]. To this, we rely on two concepts to achieve a flexible adaptation: First, an *intrinsic evolutionary search process* adapts the system to slow changes in the environment. Second, radical changes in available resources are compensated for by replacing the operational circuit with a *pre-evolved alternative* which meets the new resource constraints. To this end, we store at any time an approximated Pareto front of circuit implementations.

In this article, we first discuss models and algorithms for the evolution of hardware functions and then present the MOVES toolbox for development and simulation of EHW. Finally, we focus on two applications of EHW.

2 Models and Algorithms

In this section, we review models and algorithms for evolving hardware. In particular, we discuss cartesian genetic programs (CGP) for representing digital circuits and their extension to automated module creation and multi-objective optimization. Further, we point to the inherent trade-off between an efficient evolution and the effort required to map evolved circuits to real hardware.

2.1 Cartesian Genetic Programs

A cartesian genetic program (CGP) is a structural hardware model that arranges logic cells in a two-dimensional geometric layout [4]. Formally, a CGP model consists of $n_c \times n_r$ combinational logic blocks, n_i primary inputs, and n_o primary outputs. A logic block has n_n inputs and implements one out of n_f different logic functions of these inputs. While the primary inputs and outputs can connect to any logic block input and output, respectively, the connectivity of the logic block inputs is restricted. The input of a logic block at column c may only connect to the outputs of blocks in columns $c-l, \ldots, c-1$ as well as to the primary inputs. The levels-back parameter l restricts wiring to hardware-friendly local connections. More importantly, as only feed-forward connections are allowed, the creation of combinational feedback loops is avoided. Fig. 1a shows an example for a CGP model together with its parameters. The model in this example has five columns, four rows, four primary inputs, and two primary outputs.

Mutation is the commonly used evolutionary operator for CGP. Often formalized as a one-point operator, mutation changes a gene which encodes a node's function and input wiring with a certain probability. The straightforward implementation of a crossover operator acts on the geometrical structure of the chromosome exchanging nodes and preserving their wiring. An

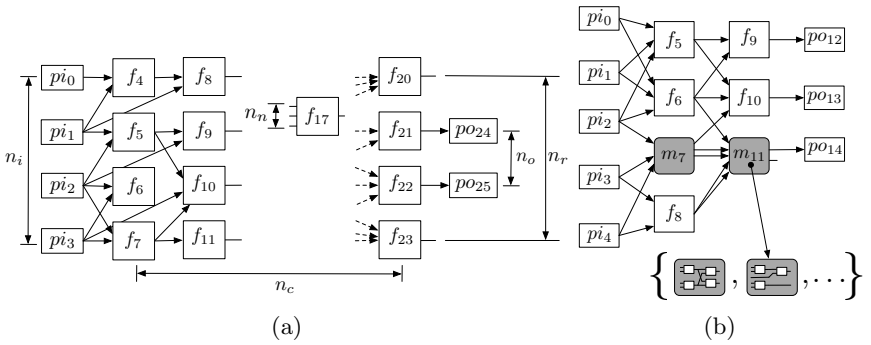

Fig. 1: Cartesian genetic programming (CGP) model, Fig. 1a, and its extension to the automatic search of reusable sub-functions, Fig. 1b.

n-point crossover divides the parents' chromosomes into $n+1$ parts based on the numbering of the nodes. The child's chromosome is then constructed by alternatively selecting partial chromosomes from the parents.

A crucial property of CGP is that nodes which do not contribute to the primary outputs remain in the chromosome and might be propagated through the generations. This property, termed *neutrality*, has been shown to improve significantly the convergence of the search process [4], as it preserves possibly useful sub-functions that can be reconnected to the genotype's active structure by few mutations.

2.2 Modular CGP

In order to improve evolvability of functions with potentially hierarchically-structured solutions, Walker et al. [5] applied the automatic search and definition for reusable sub-functions (ADF) to CGP. The methodology tries to extend a genotype's alphabet by adding more complex nodes to it. The new nodes, denominated as modules, are composed from sub-programs of candidate solutions and are subject to evolutionary selection pressure. The goal behind ADFs is the automatic search of an appropriate functional set for a distinct application. Additionally, ADF aims at the automatic increase of the functional complexity of the representation model. This allows for the evolution to act on a more complex and expressive level, thereby reducing an EA's computational effort.

Fig. 1b illustrates a potential modular CGP (MCGP) phenotype structure. Aside with the regular population of candidate solutions, MCGP propagates a set of modules through the evolution, dynamically allocating and releasing them by *compress* and *expand* operators. The probability of module creation is lower than that of module expansion, providing pressure towards modules contributing to more successful candidate solutions. When a regular mutation

operator hits a module node, it selects randomly from five different actions: the module's internal function can be mutated by the means of the standard CGP mutation operator, the input and output wiring of a module can be altered similarly to wiring mutation of a basic node. Finally, the number of inputs and outputs can be changed. For further details about MCGP we refer to Walker et al. [5] and Kaufmann and Platzner [6].

In our work, we have introduced advanced techniques for creating and propagating modules in CGP as well as presented an MCGP cross-over operator [6]. Our *age-based module creation* prefers the aggregation of primitive nodes that persisted unchanged for a large number of generations. The rationale behind this is that such "aged" nodes directly or indirectly contribute to the genotype's success and should be composed into a module. Further, we have implemented *cone-based module creation* that forms modules out of primitive nodes that form cones. Cones are a widely-used concept in circuit synthesis, especially in the area of lookup-table mapping for FPGAs, capsuling functionally related elements as a reusable entity. We have evaluated our novel techniques and compared them to the module creation approach presented in previous work. The results demonstrate the effectiveness of age-based module creation. Cone-based module creation is even more effective but only for regularly structured multiple output circuits such as multipliers. Finally, we have detailed a crossover operator that selects a cone consisting of both primitive nodes as well as modules in a source chromosome and copies this cone into a destination chromosome. Using this novel crossover operator, we are also able to apply multi-objective optimizers to MCGP as crossover allows for intra-population information exchange.

2.3 Multi-objective Optimization using CGP

The motivation for multi-objective evolutionary optimization (MOEA) of hardware lies in the desired ability of an autonomous system to react quickly to changes in the resources by instantiating an appropriately sized solution. This can be achieved by exploiting a modern, Pareto-based MOEA, which optimizes concurrently a set of mutually non-dominating solutions. To verify this concept, we have added circuit area and speed as objectives to be optimized [7]. However, multi-objective optimization of CGP circuits shows a bad convergence behavior which might be explained by the fact the modern MOEAs are targeted at global optimization. In contrast, previously known good CGP optimization algorithms are single-objective local optimizers. To successfully employ a multi-objective heuristic to CGP optimization, we have extended a state-of-the-art MOEA, SPEA2 [8], to favor a subset of objectives. The resulting algorithm performs similarly to a single-objective optimizer without showing significant negative effects on the secondary objectives. We coined the new algorithm turtle SPEA2 (TSPEA2) [7, 9]. To improve the run-time on an embedded system, we have implemented the computationally-demanding methods for preserving Pareto-front diversity as an FPGA cir-

cuit [10]. Additionally, we have developed a periodization scheme for single- and multi-objective EAs to combine global and local search techniques [11]. We have shown that periodization can significantly improve the quality of the evolved Pareto-front approximation.

2.4 Challenges of CGP

The main challenge of the CGP model when applying to real-world applications is scalability. Similarly to other EA variants, disproportion in the granularity of the representation model functional blocks and application-inherent functional granularity often results in excessive optimization effort. As CGP in its original version evolves solutions at the abstraction level of Boolean gates and single wires, it struggles to cope with applications using numerical values and the corresponding mathematical functions. In the last years, a number of approaches have been presented that address this problem. One of them is function-level evolution which uses node functions of coarser granularity and buses instead of single bit wires. We have done preliminary work on the implementation of a coarse-granular CGP using Virtex-5 DSP48E blocks as functional nodes. An algorithmic approach to improve the evolvability of CGP is to enable EAs to create hierarchical solutions by automatically seeking reusable functions. We have investigated this approach in [6].

3 Development and Simulation Tools

We have created the MOVES toolbox [7, 12, 3] for evolutionary design of digital circuits. The toolbox comprises a framework of different digital logic representation models, single and multi-objective optimizers and a set of evolutionary operators, and enables us to easily deal with the setup, control, visualization, analysis, and distribution of experiments. The overall framework is developed in Java and is, thus, platform-independent. Additionally, for experiments on embedded systems without full-fledged operating systems, some parts such as the CGP representation model and variants of single and multi-objective optimizers are also available in C without using dynamic memory allocation.

The key feature of the MOVES framework is the separation of the different functionalities required for experimenting with hardware evolution. Consequently, the modules of the framework are divided into two major groups, modules that constitute the evolutionary optimization techniques and modules that serve the experimentation process, including experiment setup and control, statistic analysis, and visualization.

An excerpt form the MOVES toolbox structure is shown in Fig. 2 and includes the hardware *representation models*, the *evolutionary algorithms*, and the *evolutionary operators*. This separation is suitable for all population-based

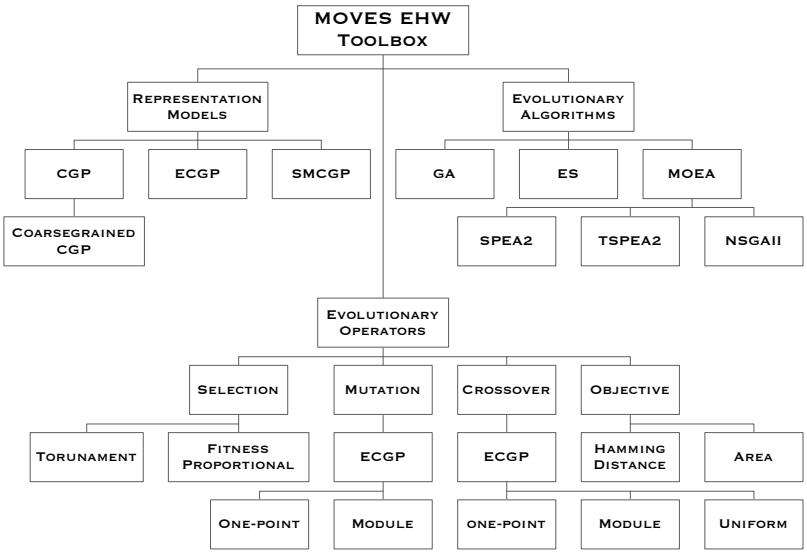

Fig. 2: Excerpt from the MOVES evolvable hardware toolbox structure.

optimization techniques that apply evolutionary operators, e.g., genetic programming, genetic algorithms, and evolutionary techniques. Within the representation models, the framework implements the regular CGP, its extension to ADFs – the MCGP, and a coarse-granular CGP model, based on Xilinx Virtex-5 DSP48E blocks as functional elements. The evolutionary algorithms comprise regular GA, $(\mu\{^{+}_{,}\}\lambda)$ ES, NSGAII, SPEA2, TSPEA2, μGA, OMOEA, and IBEA2 in the hyper-volume and the ϵ variants. The evolutionary operators can be formalized in a general form, e.g., the selection operators, or in a form specific to the representation model, e.g., the fitness evaluation, mutation, and crossover operators. Along with the known mutation and crossover operators, the MOVES framework also implements our novel cone-based and age-based selection and crossover techniques [6].

Within the MOVES framework, an experiment is defined by two human-readable configuration files; one specifies the evolutionary optimizer and another one the experiment setup. The evolutionary optimizer configuration includes the chosen representation model, evolutionary operators, and evolutionary algorithm. The experiment setup configuration comprises termination conditions, visualization settings, and the logging frequency. The configuration can be provided as regular text files or, alternatively, be entered via the framework's graphical user interface.

Experiments can be run interactively or in batch mode. In the interactive mode, the user can pause, resume, or stop the simulation at any time. Generally, the parameters controlling the experiment setup can be modified during the experiment. For example, the experiment can be executed step-wise or

continuously until some of the termination conditions are reached. The user is free to switch between these modes at any time. It is also possible to save the current search state and reload it later on to analyze it. These features are extremely useful to debug and verify new representation models and their corresponding operators, and to tune the parameters of the evolutionary algorithm. The batch mode is used for an unattended simulation. Statistical data can be gathered during the experiment and exported to a text file for later processing.

There are two visualization tools, the visualization of the evolved candidate solutions and the visualization of the evolutionary optimization process. When visualizing a genotype, the user can modify the viewed data, e.g., to manipulate input data or even change the circuit's structure and function. The standard visualization of the evolutionary optimization process displays the progress of the best fitness and the population's average fitness over the generations. This is most useful for single-objective optimizers. In experiments with multi-objective optimizers, we can display the progress of all individual fitness values, and the two-dimensional projections of the Pareto fronts. The visualizations can be done either during an experiment run or offline using previously saved experiment log streams.

For an evolvable hardware experiment, usually dozens of simulation runs are required to evaluate different parameter sets. As evolutionary algorithms are stochastic optimization methods, several runs with different random number seeds need to be conducted to derive the average behavior. Such experiments can be tedious to configure and take a very long runtime. The single experiments, however, are independent of each other and amenable to parallel execution. The MOVES framework is able to automatically create a set of experiments where parameters are varied in specified intervals and with defined step sizes, and to execute all simulations as batch jobs on a compute cluster. We employ the grid software `C`ondor that distributes the jobs on the computing nodes in the cluster, monitors the node's activities, and relocates the jobs if it becomes necessary.

4 Applications

To demonstrate our research on evolvable hardware, we select two applications covering the embedded and the high-performance-computing worlds. The first part of this section presents experiments on the classification accuracy of a run-time re-configurable FPGA pattern matching architecture [13]. At this, the focus is raised towards classification accuracy behavior and recovery, when dynamically changing the amount of on-chip resources used by the architecture. The second part shows a way to optimize the memory-to-cache address mapping function by the means of EHW. Interposing a small reconfigurable array of LUTs between the CPU's load/store unit and the cache logic allows to improve the overall execution time of a program.

4.1 Flexible Pattern Matching Architectures with EHW

The Functional Unit Row (FUR) architecture for classification tasks was first presented by Glette in [14, 15]. FUR is an architecture tailored to online evolution combined with fast reconfiguration. To facilitate online evolution, the classifier architecture can be reconfigured through partial reconfiguration and the behavior can be controlled through configuration registers. Fig. 3a shows the general organization of the classifier architecture [16, 17]. For C categories the FUR architecture consists of C Category Detection Modules (CDMs). A majority vote on the outputs of the CDMs defines the FUR architecture decision. In case of a tie, the CDM with the lower index wins. Each CDM contains M Category Classifiers (CCs), basic pattern matching elements evolved from different randomly initialized configurations and trained to detect the CDM's category. A CDM counts the number of activated CCs for a given input vector, thus the CDM output varies between 0 and M.

In our work [13], we define a single CC as a row of Functional Units (FU), shown in Fig. 3b. The FU outputs are connected to an AND gate such that in order for a CC to be activated all FU outputs have to be 1. Each FU row is evolved from an initial random bitstream, which ensures a variation in the evolved CCs.

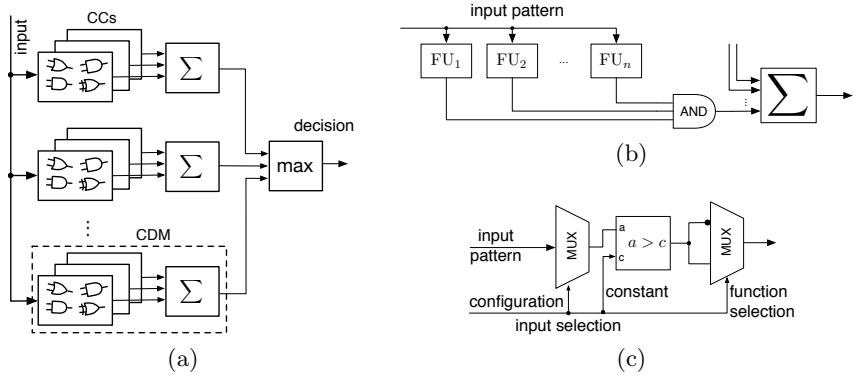

Fig. 3: The Functional Unit Row (FUR) Architecture 3a. Category Classifier (CC) 3b. Functional Unit (FU) 3c.

As depicted in Fig. 3c, an FU selects a single value from the input vector and compares it to a constant. While any number and type of functions could be imagined, Fig. 3c illustrates only two functions for clarity. Through experiments, *greater than* and *less than or equal* have shown to work well, and intuitively this allows for discriminating signals by looking at the different amplitudes.

The FUR architecture is parametrized by three values: the number of categories, FU rows in a CDM, and FUs in a FU row. We assume the numbers

of categories and FUs in a FU row as constants, reconfiguring the number of FU rows in a CDM. For a sequence $I = \{i_1, i_2, \ldots, i_k\}$ we evolve a FUR architecture having i_j FUs per CDM, then switching to i_{j+1} FUs per CDM and re-evolving the architecture without flushing the configuration evolved so far.

For our investigations we rely on the UCI machine learning repository [18] and, specifically, on the Thyroid benchmark. The Thyroid data set splits into three groups with cardinalities of 6.666, 166 and 368 samples. To evolve a FUR classifier we employ a $1 + 4$ ES scheme. With a mutation operator changing three genes in every FU row the configuration for a complete FUR architecture is evolved during a single ES run. In preparation for the experiments on the reconfigurable FUR architecture we investigate the FUR's general performance and over-fitting effects by evaluating it on a set of useful FU rows per CDM and FUs per FU row configurations. To this, we evaluate the FUR-architecture on combinations of $2, \ldots, 20$ FUs per FU row and $2, \ldots, 80$ FU rows. In these experiments, the FUR architecture shows error rates which are very close to the error rates of the best known conventional classification algorithms [13].

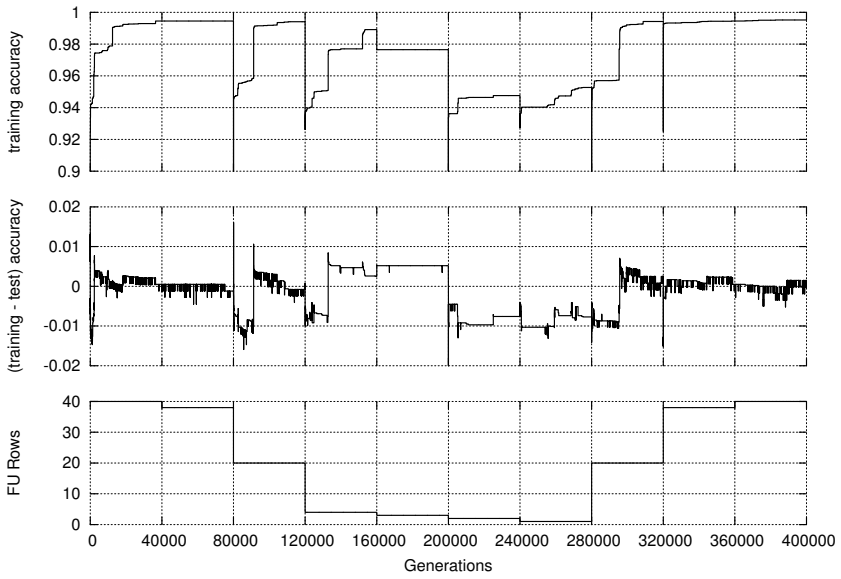

Fig. 4: Reconfigurable Thyroid benchmark: Changing the classifier's resources (number of FU rows) during the optimization run. The first diagram shows the training accuracy, the second diagram the difference between the training and test accuracy and the last diagrams shows the changes in the FU rows.

Fig. 4 visualizes the FUR's classification behavior under changes in the available resources while being under optimization. We execute a single experiment where we configure a FUR architecture with 4 FUs per FU row and change the number of FUs every 40.000 generations. We split the data set into disjoint training and test sets and start the training of the FUR classifier with 40 FU rows. Then, we gradually change the number of employed FU rows to 38, 20, 4, 3, 2, 1, 20, 30, 40 executing altogether 400.000 generations. We observe the following: the training accuracy drops significantly for almost any positive and negative change in the number of FU rows and recovers subsequently. The recovery rate of the test accuracy depends on the amount of FU rows. While for periods with few FU rows the recovery rate is slow, for periods with 20 and more FU rows the evolutionary process manages to recover the test accuracy much faster. Additionally, for configurations with very few FU rows, the test accuracy begins to deteriorate. This can be observed in Fig. 4 at generations 120.000 to 280.000.

In summary, as long as the FUR configuration contains enough FU rows, the FUR's test accuracy behavior is stable during reconfigurations. Additionally, more FU rows leverage faster convergence.

4.2 Optimizing Caches: A High-performance EHW Application

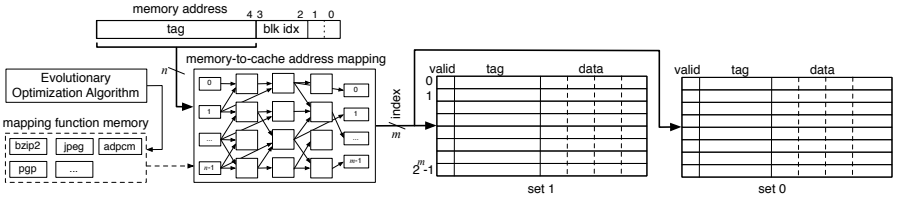

Fig. 5: The evolvable cache (EvoCache) architecture.

Cache memories are important and well-investigated elements of any modern processor's memory hierarchy. While carefully designed and balanced cache hierarchies greatly improve processor performance, they also require substantial amounts of energy. The key innovation of our evolvable cache (EvoCache) is to make the function that maps memory addresses to cache indices programmable [19].

The EvoCache approach is presented in Fig. 5. A very small reconfigurable logic fabric implements a hashing function that maps a part of a memory address to a cache line index. The hashing function is optimized to achieve a low overall execution time for a specific application. The algorithmic methods for optimization originate in the EHW domain. Our architecture provides a mapping function memory that can store several configurations for the reconfigurable logic fabric, which allows for quickly switching to different memory-

to-cache address mappings. To prevent aliasing, i.e., storing several potentially dirty copies of the same physical address at different indices in the cache, we flush the cache when a new mapping is activated.

Including EvoCaches into a processor architecture will also increase the logic area, the hit time and the overall number of memory cells for the cache. The increase in logic area is due to the reconfigurable fabric itself which is assumed to be small as the fabric comprises only a handful of look-up tables (LUTs). Additionally, we require a mapping function memory to store the configurations for the logic fabric. The size of a configuration is architecture dependent. The architecture used for this case study comes with a configuration size of 151 bytes. The increase of the cache size is due to the fact that the flexibility in the hashing function requires us to store the full address excluding block and byte offsets as tags in the cache. The additional overhead depends on the actual cache configuration. For example, a conventional 4-way set associative cache of 16 KByte data with block size of two words for a byte-addressable architecture with 32 bit addresses comes with an overhead of 25.56%, where the overhead includes for each cache block the valid bit and the tag. Switching to an EvoCache of same data size and organization increases the overhead to 34.88%. We think this overhead is bearable since today most processor designs are not restricted by silicon area but by performance and performance per energy. The increase in hit time is more critical. The additional delay depends strongly on the depth of the LUT network. This depth can be restricted in the optimization process to satisfy timing constraints. Moreover, for many embedded processors with clock frequencies well below one GHz, the pressure on the timing is moderate. High-performance processors, on the other hand, have several levels of cache where only the first level is optimized for hit time. Here, the EvoCache approach can still be applied to higher level caches.

For the experiments, we leverage our MOVES toolbox [12], which allows us to generate a set of jobs for fitness evaluation and distribute them on a compute cluster. For system simulation we rely on SimpleScalar [20] as it is easily extensible and it models a variant of the widely-used MIPS instruction set architecture. SimpleScalar allows us to establish a fine-grained energy metric, based on the memory and cache access patterns. To this, our energy estimation model splits the energy demand into a static and dynamic part. We derive the static and access energies from the CACTI cache model, the standard V58C2256 DDR SDRAM module, and a 200 MHz ARM at 90 nm, respectively.

To evaluate the EvoCache concept, we have configured a processor and its memory hierarchy in a configuration similar to those of current ARM processors [21]. The configuration includes a split first level cache and a unified second level cache. The L1 caches are 2-way associative with a hit latency of one cycle, 64 sets and a block size of 16 bytes. The L2 cache has an associativity of four ways with a hit latency of 6 cycles, 128 sets and a block size of 32 bytes. The memory bus between the L2 cache and the external memory is 8 bytes

wide. The external memory shows an access time of 18 cycles and a 2-cycle delay for consecutive data transfers in burst mode. Hence, the miss penalty for the L2 cache amounts to 24 cycles. Using this configuration, a conventional cache system for a byte-addressable architecture with 32 bit addresses has a 22 bit tag and a 6 bit index for the L1 caches and a 20 bit tag and 7 bit index for the L2 cache, respectively. For an EvoCache, the original tags and indices merge into a single tag of 28 and 27 bits for the L1 and L2 caches, respectively. We have evolved mapping functions for two optimization scenarios. In the first optimization scenario, only the first level caches (LI:I and L1:D) are EvoCaches with evolved mapping functions while in the second scenario all three caches receive evolved mapping functions. We simulate the execution of two benchmarks, `bzip2` (version 1.0.4) and `jpeg` (version 6a). For each combination of benchmark and optimization scenario, we first evolve a mapping function on a training data set. This optimization step has been repeated for 16 times to compute the average behavior. Then we evaluate EvoCaches using the best evolved `bzip2` and `jpeg` circuits on a large, diverse set of test data disjoint to the training data, This time, we additionally log the miss rates and the estimated energy consumptions.

Fig. 6 summarizes EvoCaches generalization results. The following observations can be made here: Compared to a conventional cache of equal size, EvoCaches generalize well and deliver up to 10.98% execution time improvement and up to 10.70% reduction in energy for the `bzip2` benchmark. The `jpeg` benchmark gains even higher improvements with up to 14.21% in the execution time and up to 16.43% in the energy consumption. Interestingly, `bzip2` profits from optimization of L1:I, L1:D, and L2:U caches, while `jpeg` suffers from the optimization of the second level cache. Noteworthy for the `jpeg` benchmark is the disproportion in the miss-rate gain. While the execution time improves by roughly 11% to 12%, the miss-rate gains about 35% improvement. This demonstrates that for multiple levels of cache (or sophisticated processor architectures) the total miss rate is not necessarily a suitable metric for quantitatively determining a performance improvement.

5 Conclusion

This article gives a short introduction to evolvable hardware, presents an overview over our models and tools for evolutionary digital circuit design and concludes with two case studies. In the first case study we leverage the FUR classifier architecture for creating evolvable hardware systems that can cope with fluctuating resources. We demonstrate that the FUR's generalization performance is robust to changes in the available resources as long as a certain amount of FU rows is present in the system. Furthermore, the FUR's capability to recover from a change in the available resources benefits from additional FU rows. In the second case study we present the EvoCache concept which relies on two main ideas. First, the memory-to-cache-mapping function

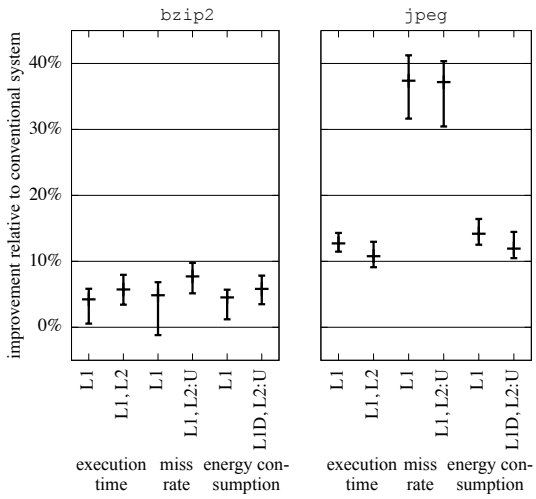

Fig. 6: EvoCache generalization performance for `bzip2` and `jpeg`.

is implemented by a small reconfigurable logic fabric. Second, the function is optimized by an evolutionary algorithm with the goal to achieve a minimal overall execution time with respect to a specific application. We evaluate the concept on the bzip2 and jpeg benchmarks. Compared to conventional caches, we observe runtime improvements of up to 10.98% for bzip2 and up to 14.31% for jpeg and energy reductions of up to 10.70% for bzip2 and up to 16.43% for jpeg.

References

1. de Garis, H.: Evolvable Hardware: Genetic Programming of a Darwin Machine. In: Intl. Conf. on Artificial Neural Nets and Genetic Algorithms. Springer (1993) 441–449 1
2. Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H., Furuya, T.: Evolving Hardware with Genetic Learning: a First Step Towards Building a Darwin Machine. In: From Animals to Animats, MIT Press (1993) 417–424 1
3. Kaufmann, P., Platzner, M.: Toward Self-adaptive Embedded Systems: Multi-objective Hardware Evolution. In: Architecture of Computing Systems (ARCS). Volume 4415 of LNCS., Springer (2007) 199–208 2, 5
4. Miller, J., Thomson, P.: Cartesian Genetic Programming. In: European Conf. on Genetic Programming (EuroGP), Springer (2000) 121–132 2, 3
5. Walker, J.A., Miller, J.F.: Evolution and Acquisition of Modules in Cartesian Genetic Programming. In: European Conf. on Genetic Programming (EuroGP). Volume 3003 of LNCS., Springer (2004) 187–197 3, 4
6. Kaufmann, P., Platzner, M.: Advanced Techniques for the Creation and Propagation of Modules in Cartesian Genetic Programming. In: Genetic and Evolutionary Computation (GECCO), ACM Press (2008) 1219 – 1226 4, 5, 6

7. Kaufmann, P., Platzner, M.: Multi-objective Intrinsic Hardware Evolution. In: Intl. Conf. Military Applications of Programmable Logic Devices (MAPLD). (2006) 4, 5
8. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, ETH Zurich (2001) 4
9. Knieper, T., Defo, B., Kaufmann, P., Platzner, M.: On Robust Evolution of Digital Hardware. In: Biologically Inspired Collaborative Computing (BICC). Volume 268 of IFIP International Federation for Information Processing., Springer (2008) 2313–222 4
10. Schumacher, T., Meiche, R., Kaufmann, P., Lübbers, E., Plessl, C., Platzner, M.: A Hardware Accelerator for k-th Nearest Neighbor Thinning. In: Proc. Intl. Conf. on Engineering of Reconfigurable Systems & Algorithms (ERSA), CSREA Press (2008) 245–251 5
11. Kaufmann, P., Knieper, T., Platzner, M.: A Novel Hybrid Evolutionary Strategy and its Periodization with Multi-objective Genetic Optimizers. In: IEEE World Congress on Computational Intelligence (WCCI), Congress on Evolutionary Computation (CEC), IEEE (2010) 541–548 5
12. Kaufmann, P., Platzner, M.: MOVES: A Modular Framework for Hardware Evolution. In: Adaptive Hardware and Systems (AHS), IEEE (2007) 447–454 5, 11
13. Knieper, T., Kaufmann, P., Glette, K., Platzner, M., Torresen, J.: Coping with Resource Fluctuations: The Run-time Reconfigurable Functional Unit Row Classifier Architecture. In: Intl. Conf. on Evolvable Systems (ICES). Volume 6274 of LNCS., Springer (2010) 250–261 7, 8, 9
14. Glette, K., Torresen, J., Yasunaga, M.: An Online EHW Pattern Recognition System Applied to Face Image Recognition. In: Applications of Evolutionary Computing (EvoWorkshops). Volume 4448 of LNCS. Springer (2007) 271–280 8
15. Torresen, J., Senland, G., Glette, K.: Partial reconfiguration applied in an online evolvable pattern recognition system. In: NORCHIP 2008, IEEE (2008) 61–64 8
16. Glette, K., Gruber, T., Kaufmann, P., Torresen, J., Sick, B., Platzner, M.: Comparing Evolvable Hardware to Conventional Classifiers for Electromyographic Prosthetic Hand Control. In: Adaptive Hardware and Systems (AHS), IEEE (2008) 32–39 8
17. Glette, K., Torresen, J., Kaufmann, P., Platzner, M.: A Comparison of Evolvable Hardware Architectures for Classification Tasks. In: Intl. Conf. on Evolvable Systems (ICES). Volume 5216 of LNCS., Springer (2008.) 22–33 8
18. Asuncion, A., Newman, D.: UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences (2007) 9
19. Kaufmann, P., Plessl, C., Platzner, M.: EvoCaches: Application-specific Adaptation of Cache Mappings. In: Adaptive Hardware and Systems (AHS), IEEE (2009) 11–18 10
20. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. In: IEEE Computer. Volume 35(2)., IEEE (2002) 59–67 11
21. ARM: ARM10E processor family. `http://www.arm.com/products/CPUs/families/ARM10EFamily.html` (2010) 11