1 Classification Hardware Evolution using Modular Approach

Many classification tasks, especially in embedded and real-time settings, require not only sufficient classification accuracy but also high performance, high energy-efficiency, and low resource usage. These objectives can be reached by a direct hardware implementation. Evolvable hardware (EHW) approaches create direct hardware implementations and have been applied successfully to classification tasks, such as the recognition of characters [12], road signs [25], faces [9], electromyographic signals [13, 6, 4], and the diagnosis of the Parkinson disease [21].

We present an EHW approach based on the EGCP hardware representation model to design classifiers for electromyographic (EMG) signals. EMG signal classification is a prerequisite for the control of prosthetic hands [4, 8]. We discuss our classifier architecture, the EMG signal domain, the hardware representation model, the evolutionary optimization technique and, finally, experimental results. The results are promising and show that EHW-based EMG signal classifiers reach a classification accuracy close to that of state-of-the-art classification techniques. Furthermore, the EWH approach lends itself to online adaptability and fault recovery, making the technology a promising candidate for future self-adaptive classifiers.

1.1 Classifier Architecture

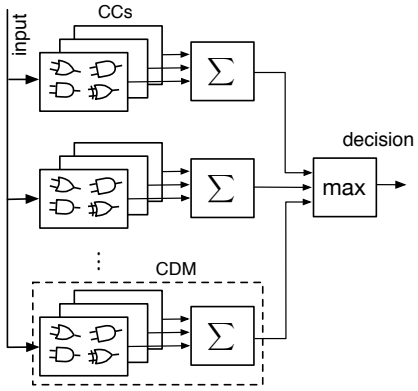

Fig. 1: EHW classifier architecture. Each category detecting module (CDM) contains a set of category classifiers (CC) that implement evolved pattern matching rules. The CDM with the highest number of satisfied rules defines the global decision.

The hardware architecture is probably the most important design issue for an EHW classifier. Figure 1 shows the structure of our classifier architecture. The basic element is the category classifier (CC) implementing an evolved pattern matching rule. A category detection module (CDM) groups several category classifiers and counts the number of satisfied rules. A global maximum detector determines the category with the most hits as classification result. In case of a draw the category with the lowest numerical index is selected.

Rather than directly evolving a huge ECGP model, all successful EHW classifiers impose a top-level structure as the one shown in Figure 1. An earlier approach uses a programmable logic array (PLA)-like structure of AND gates followed by OR gates [12]. The Increased Complexity Evolution (ICE) architecture [24] features a modular top-level structure that splits into several category detection subsystems. A similarly modular approach is the Functional Unit Row (FUR) architecture [7].

Our classifier architecture can be seen as a generalized version of previous modular architectures. All the modular architectures aim at three goals: First, the modularization of the classifier into CDMs and CCs reduces the complexity of the evolved circuits which in turn leads to a faster convergence of the optimization process. Second, the architecture scales with the number of categories by adding more CDMs. Third, classification accuracies can be increased by using multiple CCs and taking a majority vote decision at the end.

1.2 EMG Signal Domain

For EMG data acquisition, we use a measurement system comprising four components: EMG electrodes (Tyco Arbo*, Ag/AgCl, 35 mm), amplifiers (Biovison [3]), A/D converters (N.I. [17]), and a standard computer. Our system continuously monitors four sensor channels with 14 bit resolution at a sampling rate of 6 kHz. Two important requirements for such a measurement system are the reduction of noise in the analog signal domain and a reproducible biomechanical experiment setup.

To reduce noise, we employ an optical bridge (Sonowin [22]) to galvanically decouple the signal amplifiers and the A/D converters from the computer that accumulates the data. A separate battery provides a stable power supply to the amplifiers and A/D converters. Moreover, the amplifiers are placed as near as 10 cm to the skin-attached electrodes in order to minimize parasitic inductance of a significant level.

We place the four electrode pairs on the top, bottom, medial, and lateral sides of the forearm as shown in Figure 2, with the reference at the wrist. The exact electrode positions are determined specifically for each test subject to obtain pronounced signals. After this initial calibration, the electrode positions are marked to be able to re-establish the experimental setup on different days.

In a single experiment run, the test subject has to perform 20 iterations of a sequence of eight different movements. These movements are *open, close, flexion, extension, ulnar deviation, radial deviation, pronation,* and *supination*, and are de-

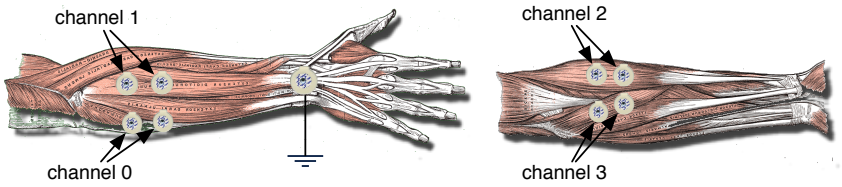

Fig. 2: Sensor placement (muscle anatomy taken from [10]).

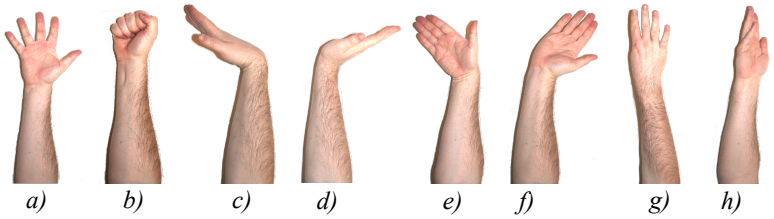

Fig. 3: Movements: *a)* open, *b)* close, *c)* flexion, *d)* extension, *e)* ulnar deviation, *f)* radial deviation, *g)* pronation and *h)* supination.

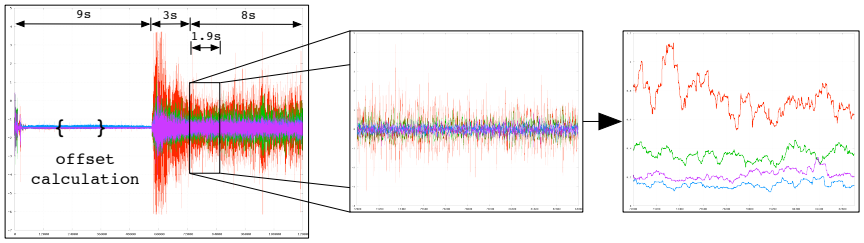

Fig. 4: EMG signal preprocessing. The left figure shows the raw signals for all four channels, consisting of a nine seconds relaxation phase, a three seconds transient phase with intensified activity, and an eight seconds steady state contraction phase. The center figure presents the DC offset-compensated first 1.9 seconds of the steady state phase, and the right figure the RMS smoothed signals from which the features are extracted.

picted in Figure 3. A single movement consists of two phases: a nine seconds relaxation and an 11 seconds contraction part. The EMG signal for the contraction part divides into a three seconds phase at the onset of the contraction containing the transient components of the EMG signal, and an eight seconds steady state phase which corresponds to a constant force contraction. A part of this steady state phase is used for classification. Figure 4 presents an example for a complete EMG signal.

Signal preprocessing and feature extraction is done completely in the digital domain. Following the approach presented by Kajitani et al. in [13], we extract the features in four steps:

1. For every channel $k, k = 1, \ldots, 4$, and movement $p, p = 1, \ldots, 8$, we calculate the sensor DC offset o_{kp} as the mean value of all signal samples between the third and the fifth second of the signal relaxation phase.
2. The steady state signal d_{ikp} is DC offset-compensated and smoothed by a root mean square (RMS) method with a window size of $w_s = 600$. The first 1.9 seconds ($11'400$ samples at 6 kHz) of the rectified and smoothed signal d'_{jkp} are calculated by
$$d'_{jkp} = \left[\frac{1}{w_s} \sum_{i=j}^{j+w_s-1} (d_{ikp} - o_{kp})^2 \right]^{\frac{1}{2}},$$
with $j = 1 \ldots 11'400$.
3. We then apply a logarithm-transformed moving average to the rectified and smoothed signal, using a window size of $w_f = 6'000$ samples and a shift amount of $s_f = 600$ samples. The non-normalized feature thus consists of 10 values and is defined as
$$f_{l_m kp} = -\log \left(\frac{1}{w_f} \sum_{j=l_m}^{l_m+w_f-1} d'_{jkp} \right),$$
with $l_m = 1 + (m-1) \cdot s_f$, and $m = 1, \ldots, 10$.
4. Finally, we normalize the features for each channel separately:
$$g_{lkp} = \frac{f_{lkp} - \min_{l,p}(f_{lkp})}{\max_{l,p}(f_{lkp}) - \min_{l,p}(f_{lkp})}$$
Taking all $k = 4$ channels into account, the feature vector for a single movement consists of 10×4 values. These 40 values are previously linearly quantized by a 1-out-of-16 encoder and fed into the classifiers. This configures the number of inputs (n_i) for the CDM's category classifiers to 640 bits.

1.3 Classifier Hardware Representation Model

We encode the category classifiers of the CDMs by ECGP chromosomes. With six category classifiers per CDM and eight categories, the complete classifier architecture is configured by a multi-chromosome genotype with 48 ECGP chromosomes. Each chromosome comprises one row ($n_r = 1$) of 50 to 250 ($n_c = 50 \ldots 250$) functional units without restriction on the length of the feed-forward wires ($l = n_c$). The functional units are either look-up-tables (LUTs) with four inputs and one output ($n_i = 4, n_o = 1$) or modules with up to ten LUTs. We do not restrict the function set for the LUTs. We apply the techniques for automatic module creation and reuse (ADF) described in Chapter **??**. However, finding category-specific patterns in input vectors modulated by a large quantum of randomness results in circuits with less structure compared to evolved circuits for arithmetic benchmarks. Based on

this observation, we solely focus on age-based module creation [15]. Table 1 lists the parameters chosen for the ECGP model as described in Chapters **??** and **??**.

Table 1: Parameters for the evolution of the ECGP EHW classifier.

n_i / n_o / n_r / n_c	640 / 1 / 1 / 50–250
n_a / n_f	4 / $\mathbb{B}^4$
#fitness evaluations per generation	4
mutation probability	1.0
mutation rate	0.03
one point mutation probability	0.6
compress / expand probability	0.1 / 0.2
module point mutation probability	0.04
add / remove module input probability	0.01 / 0.02
add / remove module output probability	0.01 / 0.02
maximum module size	10

1.4 Fitness Assignment and Evolutionary Algorithm

There are several ways to evolve the CDMs and CCs in the proposed classification architecture. Related work used incremental and two-step evolution to partition the task of evolutionary optimization into smaller subtasks. In [9] single CDMs are evolved incrementally and independently, before being assembled to a complete classifier. Two step evolution is presented in [24]. In the first step the CMDs are evolved independently. In the second step the CDMs are combined and the CDM selectors of the overall architecture are evolved considering the fitness of the complete system.

We use direct evolution of the complete classifier. Through a series of experiments we have observed that partitioning the evolutionary optimization process leads to classifiers with acceptable classification performance rather quickly, but letting the evolutionary algorithm exploit the complete search space delivers the highest classification performance.

For classifier evolution, we define the fitness as the reciprocal and squared classification error. With categories C_p $(p = 1,\ldots,P)$ and training vector set X $(X = \dot{\cup}_{i=1}^{P} X_i)$ where X_i is the set of training vector for category i, the fitness of a classifier c is defined as:

$$f(c) = \left[1 + \frac{1}{|X|}\sum_{i=1}^{P}\left[\sum_{x\in X_i}|i - c(x)|\right]^2\right]^{-1}.$$

We employ a (1+4) Evolutionary Strategy (ES) to evolve the classifier. The evolutionary scheme creates four off-springs by mutation and selects the fittest child to

become the parent for the next generation. If no child has a fitness superior to the parent's one, the parent proceeds to the next generation.

1.5 Experiments and Results

We compare the performance of our classification architecture to the following popular and state-of-the-art classification algorithms: k-th Nearest Neighbour (kNN), Decision Trees (DT), Artificial Neuronal Networks (ANN), and Support Vector Machines (SVM). We use k-fold cross validation to determine the classification rates for all classifiers. k-fold cross validation segments an overall data set into k subsets of approximately equal size. In k runs, one subset is used for testing whereas the others are used for the training of the classifiers.

We report on two experiments. The objective of the first experiment (*Day1-3*) is the investigation of the asymptotical classification performance. The collected EMG data consisting of overall 60 repetitions of eight movements recorded on three consecutive days is evaluated using the leave-one-out validation scheme, which sets k to the number of feature vectors. The second experiment (*2of3*) is defined from the application perspective: An amputee would rely on data from past days to train the prosthesis for use on the next day. Thus, we use 3-fold cross validation with data sets defined by the three recording days.

To compare the classification performance of the different approaches we use the classification accuracy expressed by the error rate. Table 2(a) presents the training error rates pointing to the classifiers' approximation abilities, and Table 2(b) presents the test error rates showing the classifiers' generalization abilities. Table 3 presents the test error rates for the individual movements. In all tables, EHW denotes our classification architecture and the best performing classifier is marked in bold.

Since the EHW classifier is evolved from random genomes, each classifier implements a different combinational function and the classification rates vary slightly. For the *Day1–3* experiment, the leave-one-out technique requires us to evolve a rather high number of classifiers which averages out the differences in initial genomes. The *2of3* experiment, however, generates only three classifiers. To achieve sound error rates, we evolve 10×3 classifiers and average the results. Further, the training error for EHW is 5% in all experiments as we use this threshold as termination criterion.

The comparison of experiments *Day1–3* and *2of3* in Table 2 shows that almost all algorithms achieve better training but worse test results for the *2of3* experiment. This is due to the fact that for the *2of3* experiment the training set is much smaller allowing for tighter approximation, while the verification set is larger compared to the *Day1–3* experiment.

The best performing algorithms in both experiments are kNN, ANN and SVM with only marginal performance differences, followed by EHW and DT. Interestingly, the good performance of the simple kNN techniques points to the fact that our

Table 2: Training errors (approximation) are summarized in (a), test errors (generalization) are summarized in (b).

(a)

	Day1–3	*2of3*
*k*NN	3.80 %	3.74 %
DT	2.68 %	2.69 %
ANN	**0.22%**	**0.11%**
SVM	3.58 %	3.03 %
EHW	5.00 %	5.00 %

(b)

	Day1–3	*2of3*
*k*NN	4.25 %	**5.61** %
DT	8.29 %	13.26 %
ANN	**2.70** %	6.02 %
SVM	3.80 %	6.51 %
EHW	9.00 %	10.6 %

Table 3: 2of3: Individual movement errors (generalization).

	Close	Extension	Flexion	Open	Pronation	Radial dev.	Supination	Ulnar dev.
KNN	**3.92%**	**0.0%**	**0.0%**	5.17%	6.78%	19.23%	**3.57%**	6.78%
DT	5.88%	6.78%	5.66%	13.79%	16.95%	17.31%	39.29%	15.25%
ANN	9.80%	1.69%	**0.0%**	8.62%	**5.08%**	15.38%	5.36%	**3.39%**
SVM	5.88%	**0.0%**	**0.0%**	8.62%	8.47%	19.23%	**3.57%**	6.78%
EHW	5.90%	12.20%	**0.0%**	**3.50%**	12.20%	**12.80%**	22.20%	5.80%

EMG signal classification problem is not too hard. Table 3 shows, however, that the individual movements are not consistently classified best by *k*NN, ANN and SVM. For the 'flexion', 'open' and 'radial deviation' movements the EHW classifier is on par or even outperforms *k*NN, ANN and SVM.

The main result of our EMG signal classification experiments is that for prosthesis control EHW classifiers achieve accuracies sufficiently close to that of state-of-the-art classification algorithms [11]. Depending on the specific set of movements to classify, other algorithms might deliver slightly increased accuracies. The appeal of EHW classifiers, however, roots in their compactness, fast computation, and their suitability for self-adaptation.

2 EvoCaches: Application-specific Adaptation of Cache Mappings

In this section we present EvoCache, a novel approach for implementing application-specific caches. The key innovation of EvoCache is to make the function that maps memory addresses from the CPU address space to cache indices programmable. We support arbitrary Boolean mapping functions that are implemented within a small reconfigurable logic fabric. For finding suitable cache mapping functions we rely on Cartesian Genetic Programming for circuit representation and Evolutionary Strategies for fast optimization. We evaluate the use of EvoCache in an embedded processor for two specific applications (JPEG and BZIP2 compression) with respect to execution time, cache miss rate and energy consumption. We show that the evolv-

able hardware approach for optimizing the cache functions not only significantly improves the cache performance for the training data used during optimization, but that the evolved mapping functions generalize very well. Compared to a conventional cache architecture, EvoCache applied to test data achieves a reduction in execution time of up to 14.31% for JPEG (10.98% for BZIP2), and in energy consumption by 16.43% for JPEG (10.70% for BZIP2). We also discuss the integration of EvoCache into the operating system and show that the area and delay overheads introduced by EvoCache are acceptable.

2.1 The EvoCache Concept

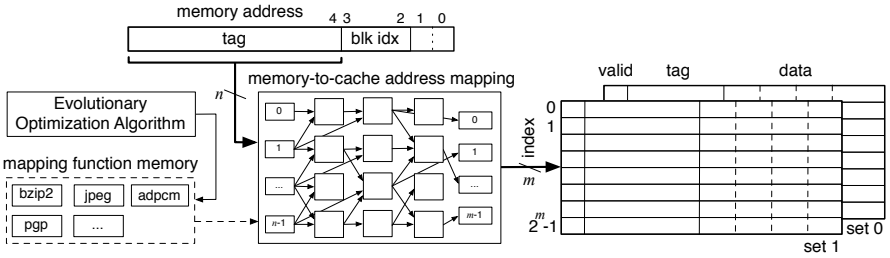

Fig. 5: The evolvable cache (EvoCache) architecture provides a configurable mapping from CPU memory addresses to cache indices. The optimization process reconfigures node functions and the wiring between the nodes. The nodes represent Boolean functions with n_a inputs. The figure shows an example of a two way set associative cache [16].

The key idea of the EvoCache approach is presented in Figure 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 Evolvable Hardware (EHW) domain which aims at automated circuit design and optimization by combining evolutionary algorithms and reconfigurable hardware technology.

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.

The EvoCache approach is orthogonal to other work trying to select and/or reconfigure the cache organization in an application-specific way, e.g., [1, 18, 31]. While Figure 5 displays an address mapping for a byte-addressable architecture to a

2-way associative cache with a block size of four words, the EvoCache principle is applicable to all possible configurations and levels of caches. Compared to classical modulo mappings or mappings based on bit permutations [23] and XOR functions [27], EvoCaches utilize more complex, evolved hashing functions allowing them to reduce an application's overall execution time and energy requirement as we will show in the remainder of this section.

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 the case study in this paper 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 incurred 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.

Integrating EvoCaches with a standard operating system environment requires only a few modifications. For keeping the information about the cache mapping as close as possible with the application's binary, we choose to store it as an optional section in the binary itself. Since all commonly used binary formats, such as COFF, ELF or MachO, support storing multiple code and data areas (sections) in the binary, this feature can be easily added without requiring a new binary format. The cache mapping information can be added by the standard linker. Since this information is small (typically a few hundred bits) the binary size is only slightly increased.

For activating the cache mapping when an application is started, the application loader needs to be extended. After loading the application's text and data sections, the loader configures the mapping function memory according to the information stored in the binary. The operating system also stores the cache mapping as part of the context of a process. For multi-tasking operating systems, the operating system changes the cache mapping at every context switch to a user task.

The proposed change of the binary format integrates the support for EvoCaches in a backward compatible way. First, the additional section containing the cache

mapping will be ignored when the application is executed on a system without EvoCache. Second, systems with EvoCache can still execute standard binaries. If the loader detects that no cache mapping information is present, it will initialize the classical modulo cache mapping.

In next sections, we will determine a suitable cache mapping function for an application and a specific input data set with the evolutionary optimizer, and then evaluate the performance on different input data sets to verify the generalization capability of EvoCaches.

2.2 System Simulation and Metrics

This section describes the configuration of the hardware representation model, the evolutionary optimization algorithm, and the method used to evaluate the fitness of candidate circuits.

We have configured the CGP model to use look-up tables (LUTs) with four inputs ($n_a = 4$) as node functions. The functional set f for the nodes has not been constrained, i.e., $f = \mathbb{B}^{16}$. To reduce the search space and thus increase the efficacy of the evolutionary optimizer, we configure the CGP model to have only one row ($n_r = 1$) but $n_c = 32$ columns. The levels-back parameter is set to $l = 31$. The circuit's inputs are fed from $n_i = 27$ primary inputs taken from the memory address. The $n_o = 15$ bit outputs of the circuit encode the cache line index. The circuit depth is an important parameter for EvoCaches as it is proportional to the delay of the resulting hashing function which adds to the cache hit time. While constraining the circuit depth during optimization can be easily done, the experiments have been conducted with unconstrained circuit depth. Instead, in Section 2.3 we report on the depths and sizes of the evolved circuits.

As optimization technique we use an $1+4$ ES scheme, where in every generation one parent creates four children through mutation. One of the fittest children proceeds to the next generation. The parent is promoted to the next generation if it excels all children. The mutation operator modifies a single gene during child creation, i.e., the function of a single logic node or the wiring of one of its inputs is affected.

2.2.1 Fitness Evaluation and SimpleScalar Integration

For the experiments, we leverage our MOVES EHW toolbox [14], which comprises different hardware representation models and evolutionary optimizers. Additionally, the toolbox generates a set of jobs for fitness evaluation and distributes them on a compute cluster.

The tool setup is presented in Figure 6. The MOVES toolbox includes the CGP model and the ES. Whenever a new candidate circuit is generated, it is passed to the processor simulator SimpleScalar [2] for fitness evaluation. SimpleScalar reads the

description of the circuit and simulates the execution of a specified benchmark and input data on a processor with given cache configuration in a cycle-accurate manner. We have chosen SimpleScalar for system simulation as it is easily extensible and it models a variant of the widely-used MIPS instruction set architecture. Two modifications to the original SimpleScalar tool have been necessary. First, its command line interface has been extended to include the activation and specification of up to four mapping functions. These circuit specifications are read in and stored in a data structure. Second, for the actual mapping between addresses and cache line index, SimpleScalar needs to determine the logic result for the mapping function. To this end, the circuit evaluation routine already available in the MOVES toolbox has been extracted into a library (moves.lib) and linked with SimpleScalar. In each simulation run, SimpleScalar determines an application's overall runtime and feeds it back as fitness value into the evolutionary optimizer.

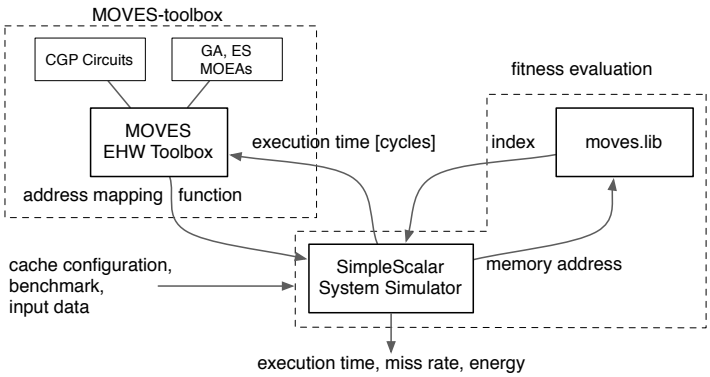

Fig. 6: EvoCache tool setup: SimpleScalar is invoked by the MOVES toolbox and returns the overall execution time in clock cycles as a fitness measure.

2.2.2 Miss Rate and Energy

Besides the cycle-accurate runtime, SimpleScalar determines the miss rates for the different levels of caches. Our interest in the miss rates is motivated by the fact that related work used miss rates to measure the fitness of a specific cache configuration. However, for more sophisticated processor architectures metrics solely based on miss rates might be less conclusive than execution time. The downside of using the cycle-accurate execution time as main metric is the long simulation time. We have constrained the simulation time to three to five minutes for a single fitness evaluation, which results in a overall runtime of roughly one week for a single and complete ES run. These constraints on the simulation time resulted in limiting the input data size for the benchmarked applications to some 100 KBytes which poses sufficient pressure on the cache architecture of an embedded proces-

sor as modeled in our work. However, a modern general-purpose processor's cache architecture would not be stressed sufficiently and thus require the simulation of application runs on larger data sizes.

As energy estimate we use a variant of the energy model presented in [31] which splits the energy demand in a static and dynamic part. We model an embedded processor with up to two levels of cache and an external memory. For each of the caches, i.e., split level one caches L1:I and L1:D and unified level two cache L2:U as well as for the external memory, the static or stand-by energy per cycle is given by $E_{L1:I,s}, E_{L1:D,s}, E_{L2:U,s}$ and $E_{M,s}$. With c as the number of clock cycles required for program execution, the static energy is

$$E_{static} = (E_{L1:I,s} + E_{L1:D,s} + E_{L2:U,s} + E_{M,s}) \cdot c$$

The dynamic energies per access are given by $E_{L1:I,d}, E_{L1:D,d}, E_{L2:U,d}$ and $E_{M,d}$ and the number of accesses as $a_{L1:I}, a_{L1:D}, a_{L2:U}$ and a_M. Thus, the dynamic energy results in

$$\begin{aligned} E_{dynamic} = {} & E_{L1:I,d} \cdot a_{L1:I} + E_{L1:D,d} \cdot a_{L1:D} + \\ & E_{L2:U,d} \cdot a_{L2:U} + E_{M,d} \cdot a_M \end{aligned}$$

The actual values in $[nJ]$ for the static energy per cycle and dynamic energy per access are derived from the CACTI cache model [20] for a 90 nm technology node. For the external memory, these values have been derived from the data-sheet of a standard V58C2256 DDR SDRAM module. The overall number of clock cycles and the number of accesses are determined by the SimpleScalar simulator. Finally, the CPU energy E_{cpu} is computed by assuming a CPU with an average power consumption of 0.45 mW per MHz at a clock frequency of 200 MHz implemented in 90 nm technology [29]. The overall energy for an application run thus adds up to

$$E = E_{cpu} + E_{static} + E_{dynamic}$$

2.3 Experiments and Results

To evaluate the EvoCache concept, we have configured a processor and its memory hierarchy in a configuration similar to those of current ARM processors [29]. The configuration is shown in Figure 7 and 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. Thus, a single chromosome describing the system's mapping functions consists of two CGP chromosomes in the first optimization scenario and of three CGP chromosomes in the second optimization scenario.

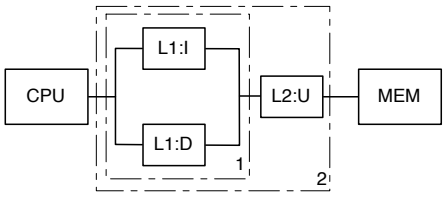

Fig. 7: Two memory hierarchy configurations considered for the optimization of the address mapping function: (1) optimization of split first level caches (L1:I,L1:D), (2) optimization of an additional unified second level cache (L1:I, L1:D, L2:U).

For evaluation we have simulated the execution of two benchmarks, `BZIP2` (version 1.0.4) and `JPEG` (version 6a), each with different sets of input data. `BZIP2` is a recent data compressor based on the Burrows-Wheeler transformation [19] and was reported to cause a large amount of cache misses. The picture encoding application `JPEG` [28] is a commonly used benchmark for performance analysis.

For each combination of benchmark and optimization scenario, we have proceeded as follows. First, we have evolved a mapping function for a given input data set, denoted as training data. This optimization step has been repeated for 16 times. To study the potential of EvoCaches, we analyze the fitness development of the best and the worst individual in each generation as well as the average over all 16 runs over two reference systems. These are a cache-less system with a one cycle memory access time which as such is unrealistic but serves as point of reference, and a two-level cache with classical modulo address mapping functions.

Second, we have determined the generalization behavior by evaluating the best evolved mapping functions on different sets of input data, denoted as test data. These results are actually more important than the results achieved for training data, as they reflect the practical use case of EvoCaches.

For `BZIP2`, the training data set consists of the HTML code from Wikipedia's page on 'Genetic Programming' [30]. The test data consists of 30 data sets partitioned in HTML data, Linux binaries, and human-readable text files. For `JPEG`, the training data set originates from the standard picture contained in the `JPEG` source code distribution. As test data, we use ten data sets from [26] and [5].

2.3.1 Training EvoCaches

Table 4 summarizes the training results. The numbers in the 'absolute'-columns are calculated as reciprocal of the overall execution time relative to the cache-less reference system with a one cycle access time. That is, since `BZIP2` executing training data on the cache-less reference system requires 13'131'325 cycles and on the classical modulo cache 34'417'080 cycles (2.62X slowdown), the modulo cache rates at 0.3815. Analog, the `JPEG` benchmark is executed on a one cycle access memory system in 47'723'253 cycles, and in 80'148'296 cycles on a classical modulo cache system. It is a slow-down of 1.67X and rate the `JPEG` benchmark at 0.5954. In the 'relative'-column noted ratio is the speed-up to the modulo cache system. We can observe that the average performance achieved for optimizing both cache levels is actually higher than the performance achieved for optimizing only level one caches. The best individual with 17.1% improvement in runtime is found, however, by optimizing level one caches.

		BZIP2		JPEG	
		absolute	relative	absolute	relative
modulo cache		0.3815	-	0.5954	-
L1:I,D	avg	0.4038	5.8%	0.6623	11.2%
	max	0.4086	7.1%	0.6975	17.1%
L1:I,D,	avg	0.4037	5.8%	0.6718	12.8%
L2:U	max	0.4174	9.4%	0.6962	16.9%

Table 4: Performance of the average and the best individuals on training data for **BZIP2** after 2500 generations and for **JPEG** after 1400 generations

2.3.2 Testing EvoCaches

To verify the generalization performance of EvoCaches, we have evaluated the execution times, the miss rates, and the energy requirements for `BZIP2` and `JPEG` and the different optimizations scenarios. The test data for `BZIP2` comprise ten data sets taken from Linux binaries (ELF benchmark), ten data sets taken from `HTML` dumps of popular web sites (HTML benchmark), and ten data sets taken from RFCs (TXT benchmark). The detailed results are shown in Table 5. In this table, the optimization scenario L1 denotes optimization of level one caches, L12 the optimization of both levels of caches. The numbers for a single benchmark and optimization scenario are averaged over the according ten data sets and measured relatively to the performance of a conventional system with modulo address mappings. That is, positive percentages indicate an improvement in execution time, a reduction in miss rate, and a reduction in energy. The miss rates for all caches have been added to achieve the miss rate metric.

		BZIP2						JPEG	
		ELF		HTML		TXT			
		L1	L12	L1	L12	L1	L12	L1	L12
execution time	best	4.92%	5.90%	5.31%	6.98%	7.30%	10.98%	14.31%	12.96%
	average	4.36%	5.60%	3.86%	4.94%	4.49%	6.66%	12.73%	10.78%
	worst	3.36%	4.87%	2.47%	3.46%	-4.15%	1.95%	11.48%	9.12%
miss rate	best	6.11%	9.00%	5.94%	8.92%	8.45%	11.38%	41.25%	40.35%
	average	5.59%	8.51%	4.15%	6.41%	4.82%	8.26%	37.40%	37.19%
	worst	4.13%	7.94%	1.64%	4.88%	-9.31%	2.63%	31.64%	30.46%
energy requirement	best	4.64%	5.49%	5.61%	7.31%	6.88%	10.70%	16.43%	14.46%
	average	4.13%	5.23%	4.53%	5.29%	4.93%	6.98%	14.19%	11.93%
	worst	3.27%	4.57%	3.21%	3.77%	2.83%	2.16%	12.53%	10.49%

Table 5: EvoCache generalization performance for `BZIP2` trained on the Wikipedia Genetic Programming (GP) HTML page. The test data are partitioned into compressing Linux binaries in ELF format (`bash`, `cpio`, `dbus-daemon`, `awk`, `sh`, `gawk`, `tar`, `tcsh`, `vim`, `zsh`), web pages in HTML format (`Ancient Egypt [W]`, `Ancient Greece [W]`, `Ancient Rome [W]`, `Germany [W]`, `heise.de`, `Andrey Kolmogorov [W]`, `sailinganarchy.com`, `spiegel.de`, `wired.com`, `slashdot.org`) and text files (`rfc 2068`, `2246`, `845`, `1000`, `1001`, `1002`, `1005`, `1008`, `1009`, `2658`). Data sets marked with [W] have been collected from wikipedia.org.

The following observations can be made for the `BZIP2` benchmark analyzing the results in Table 5:

- EvoCaches generalize well and deliver for all test data substantial performance improvements. The improvements in execution time are up to 10.98% and the reductions in energy are up to 10.70%.
- Having EvoCaches in both levels of cache (L1:I, L1:D and L2:U) leads to higher performance gains than having EvoCaches only in level one.

For testing EvoCaches on `JPEG`, we have selected ten images from [26] and [5]. The detailed results are shown in Table 5 and can be summarized as follows:

- EvoCaches again generalize well with even larger improvements in execution time (up to 14.31%) and reductions in energy (up to 16.43%).
- The average performance when optimizing L1 caches only is about 2% higher than when optimizing both cache levels. This corresponds with the observation made when training EvoCaches for `JPEG` where the best training performance was reached by optimizing L1 caches only. Consequently, the individual with best test performance gains better test performance, even if not being optimized additionally for the L2 cache.
- While the reductions in the miss rates are rather high, the reductions in execution times are lower. 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.

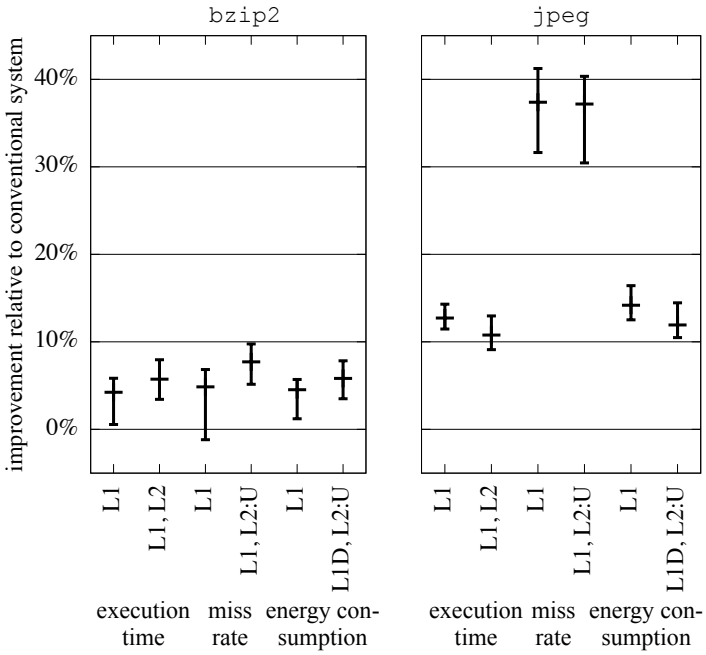

Fig. 8: Summary of the EvoCache generalization performance for BZIP2 and JPEG. The data is for randomly initialized mapping functions. The best, worst and average values are indicated for every optimization scenario and metric.

		BZIP2		JPEG	
		delay	size	delay	size
L1:I,D	avg	4.63	16.94	4.38	15.13
	max	7	22	7	22
	best	4	18	6	19
L1:I,D	avg	4.19	15.81	4.44	15.44
L2:U	max	8	24	6	22
	best	3	14	4	17

Table 6: Area and delay parameters for the evolved reconfigurable address mapping functions.

The results of our experiments are summarized in Figure 8. The figure shows for both benchmarks, BZIP2 and JPEG, and optimization scenarios the relative improvement for EvoCaches in execution time, miss rate and energy requirement over a modulo address mapping function.

The area (number of 4-LUTs) and the delay (depth of the circuit) parameters for the resulting reconfigurable logic circuits are presented in Table 6. Besides the average and maximal values, also the values for the fittest circuit which has been

used for testing is listed. These circuits show depths between three and six LUTs. It has to be noted that the circuits resulted from an evolutionary design process and have thus not been optimized for area or delay. Delay minimization could possibly further reduce the circuit's propagation time and thus the cache hit time.

2.4 Conclusion

In this section, we have presented EvoCaches that rely on two main ideas. First, the function mapping an address to a cache line index 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 have defined different optimization scenarios, optimizing split level one caches and, additionally, a unified level two cache and conducted experiments with `BZIP2` and `JPEG` benchmarks. After evolving the mapping functions, we have tested the best solutions on independent data sets and evaluated the overall execution times, miss rates, and energy requirements. Compared to conventional caches, we have observed 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`.

Index

References

1. D. H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.
2. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.
3. Biovision. EMG Amplifier. `www.biovison.eu`.
4. A. Boschmann, P. Kaufmann, M. Platzner, and M. Winkler. Towards Multi-movement Hand Prostheses: Combining Adaptive Classification with High Precision Sockets. In *Proceedings of the 2nd European Conference on Technically Assisted Rehabilitation (TAR'09)*, Berlin, Germany, 2009.
5. Classis test still images. http://hlevkin.com.
6. K. Glette, T. Gruber, P. Kaufmann, J. Torresen, B. Sick, and M. Platzner. Comparing Evolvable Hardware to Conventional Classifiers for Electromyographic Prosthetic Hand Control. In *Proceedings 3nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS'08)*, pages 32–39. IEEE Computer Society, 2008.
7. K. Glette and J. Torresen. A flexible on-chip evolution system implemented on a xilinx virtex-ii pro device. In *Evolvable Systems: From Biology to Hardware (ICES)*, volume 3637 of *LNCS*, pages 66–75. Springer Berlin / Heidelberg, 2005.
8. K. Glette, J. Torresen, P. Kaufmann, and M. Platzner. A Comparison of Evolvable Hardware Architectures for Classification Tasks. In *Evolvable Systems: From Biology to Hardware (ICES'08)*, LNCS. Springer, March 2008.
9. K. Glette, J. Torresen, and M. Yasunaga. An Online EHW Pattern Recognition System Applied to Face Image Recognition. In *Applications of Evolutionary Computing*, volume 4448 of *LNCS*, pages 271–280. Springer Berlin / Heidelberg, 2007.
10. H. Gray. Anatomy of the Human Body. 1918. Retrieved from Wikimedia Commons.
11. L. Hargrove, Y. Losier, B. Lock, K. Englehart, and B. Hudgins. A real-time pattern recognition based myoelectric control usability study implemented in a virtual environment. In *Engineering in Medicine and Biology Society (EMBS)*. IEEE, August 2007.
12. T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, B. Manderick, and T. Furuya. Evolvable Hardware and its Applications to Pattern Recognition and Fault-Tolerant Systems. In *Towards Evolvable Hardware: The evolutionary Engineering Approach*, volume 1062 of *LNCS*, pages 118–135. Springer, April 1996.
13. I. Kajitani, I. Sekita, N. Otsu, and T. Higuchi. Improvements to the Action Decision Rate for a Multi-Function Prosthetic Hand. In *Proceedings 1st International Symposium on Measurement, Analysis and Modeling of Human Functions (ISHF)*, pages 84–89, 2001.
14. P. Kaufmann and M. Platzner. MOVES: A modular framework for hardware evolution. In *Proc. NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 447–454. IEEE, 2007.
15. P. Kaufmann and M. Platzner. Advanced Techniques for the Creation and Propagation of Modules in Cartesian Genetic Programming. In *Proceedings 10th Conference on Genetic and Evolutionary Eomputation (GECCO'08)*, pages 1219 – 1226. ACM Press, 2008.
16. P. Kaufmann, C. Plessl, and M. Platzner. EvoCaches: Application-specific Adaptation of Cache Mappings. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 11–18, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
17. National Instruments. USB-6009. `www.ni.com`.
18. P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. *Proc. Int. Symp. on Computer Architecture (ISCA)*, 28(2):214–224, 2000.
19. J. Seward. bzip2: A Freely Available, Patent Free, High-quality Data Compressor, 2009.
20. P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, COMPAQ Western Research Lab, Palo Alto, California 94301 USA, 1999.

21. S. L. Smith, A. J. Greensted, and J. Timmis. Hardware Acceleration of an Immune Network Inspired Evolutionary Algorithm for Medical Diagnosis. In *Evolvable Systems: From Biology to Hardware (ICES)*, volume 5216 of *LNCS*, pages 34–46. Springer Berlin / Heidelberg, 2008.
22. Sonowin. USI-01 USB Isolator. `www.sonowin.de`.
23. M. Stanca, S. Vassiliadis, S. Cotofana, and H. Corporaal. Hashed Addressed Caches for Embedded Pointer Based Codes. In *Proc. Int. Conf. on Parallel Processing (Euro-Par)*, pages 965–968. Springer, 2000.
24. J. Torresen. Increased Complexity Evolution Applied to Evolvable Hardware. In *Smart Engineering System Design: Neural Networks, Fuzzy Logic , Evolutionary Programming, Data Mining, and Complex Systems (ANNIE)*, pages 429–436. ASME Press, 1999.
25. J. Torresen. Scalable Evolvable Hardware Applied to Road Image Recognition. In *EH '00: Proceedings of the 2nd NASA/DoD workshop on Evolvable Hardware*, page 245, Washington, DC, USA, 2000. IEEE Computer Society.
26. The USC-SIPI image database.
27. H. Vandierendonck and K. D. Bosschere. Constructing optimal XOR-functions to minimize cache conflict misses. In *Proc. Int. Conf. on Architecture of Computing Systems (ARCS)*, pages 261–272. Springer, 2008.
28. G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.
29. ARM10E processor family. http://www.arm.com/products/CPUs/families/ARM10EFamily.html.
30. Genetic programming. http://en.wikipedia.org/wiki/Genetic_programming.
31. C. Zhang, F. Vahid, and R. Lysecky. A Self-tuning Cache Architecture for Embedded Systems. *Trans. on Embedded Computing Systems*, 3(2):407–425, 2004.