EvoCaches: Application-specific Adaptation of Cache Mappings

Paul Kaufmann, Christian Plessl, Marco Platzner
University of Paderborn, Germany
{paul.kaufmann, christian.plessl, platzner}@uni-paderborn.de

***Abstract*—In this work 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 techniques from the evolvable hardware domain and utilize an evolutionary optimization procedure. 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 evolvable 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.**

I. Introduction and Related Work

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. In order to explore the performance energy trade-off for caches, Albonesi [1] proposed so-called selective cache ways. Selective cache ways is a technique to switch off certain ways of a k-way set associative cache to save energy in times of modest cache activity. The implementation foresees a new control register, the cache way select register, set by software using special instructions to signal the hardware which cache ways to enable or disable. Albonesi details the hardware organization of selective cache ways and evaluates performance and energy on CPU benchmarks including applications of the SPEC95 benchmark suite. For a 4-way set associative cache he reports a 40% energy reduction with only 2% performance degradation. The work points to software profiling tools and on-chip performance counters that could be used to drive the decision on the actually used number of cache ways but does not propose a specific method for finding the optimal cache configuration.

Ranganathan et al. [2] investigate reconfigurable caches tailored to application domains, in particular to CPU bound media processing. They propose to use the available cache memory for different purposes or modes, such as table lookup for instruction reuse, software and hardware data prefetching, or compiler and application controlled data memory. The implementation of the reconfigurable cache relies on a proper partitioning of the cache memory and is conceptually similar to Albonesi's approach [1]. The authors envision only two to three different cache configurations which are built-in at design time. At run-time, switching between these configurations would be controlled by a few multiplexers. For the instruction reuse mode, experiments with media processing benchmarks demonstrate improvements in IPC (instructions per cycle) of 4%–20%. The reconfiguration process and its triggering are not detailed, however. The paper outlines that a cache reconfiguration could take place at different frequencies, e.g., when a new application starts or even at the beginning of a new loop, and can be controlled by hardware or software.

Zhang et al. [3] present self-tuning cache architectures for embedded processors. They assume a cache with several dynamically reconfigurable parameters including the cache size, the associativity, the block size, and the way prediction. Each parameter can be set to three values except the way prediction which is a binary. The authors propose to augment the processor with a hardware tuner module that adaptively selects a cache configuration at run-time. The algorithm implemented by the tuner is a rather simple search heuristic relying on an experimentally determined ordering of parameters. Using an energy estimate based on the static and dynamic energies of the CPU, cache and external memory, savings of up to 40% in energy required for total memory accesses are reported.

All approaches discussed so far try to modify the cache configuration, i.e., the active cache size, the associativity, and the block size, or more generally the use of the on-chip memory cells. A radically different and less-investigated approach is to modify the translation or mapping between address bits and cache line index. Classically, the lower address bits are used for byte and block offsets in case of a byte-addressable architecture and multi-word cache blocks, respectively. The remaining higher address bits are split into cache index and tag. The cache index is then binary decoded to select the cache line which effectively results in a modulo address mapping.

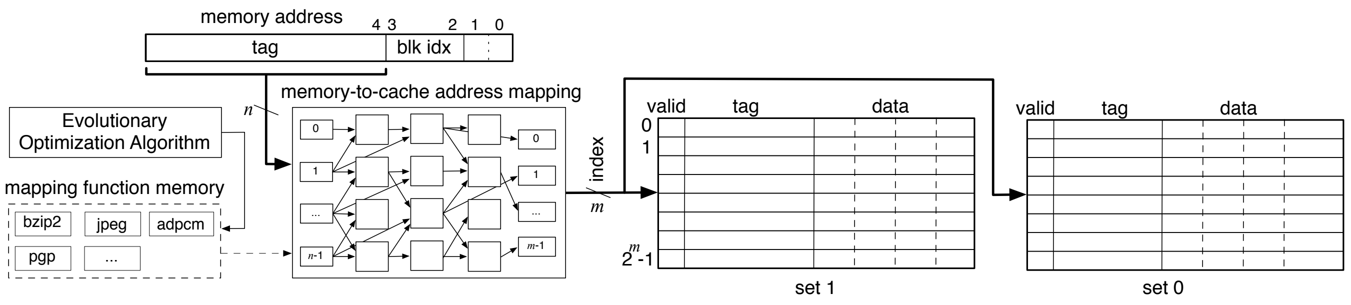


Figure 1. The evolvable cache (EvoCache) architecture provides a configurable mapping from CPU memory addresses to cache indices. The optimization process reconfigures node functions and the wiring between the nodes. The nodes represent Boolean functions with n_n inputs (see Figure 2). The figure shows an example of a two way set associative cache.

The mapping from address to cache line index can be seen as hashing problem and there a many possible hashing functions besides the typical modulo mapping. Stanca et al. [4] investigate a hashed address mapping, the so-called bit juggling technique, that permutes bits from the original address to form the new index; the tag however remains untouched. Such an address mapping is suspected to work well for pointer-based code that shows very low spatial and temporal locality. The authors experiment with level one data caches (L1:D) and report reductions in miss rates of up to 12% for small caches and an MPEG encoder/decoder benchmark. The paper does, however, neither discuss the adaptation of the address mapping nor techniques to determine suitable permutations.

More recent work [5], [6] on application-specific reconfigurable cache hashing functions focuses on XOR functions where each index bit is the XOR of exactly two address bits. The restriction to two input bits is motivated by the simplicity of the resulting logic. Heuristic and optimal algorithms are discussed for determining the input bits depending on the memory access stream of a specific application run, where optimality refers to a minimum number of cache misses. In [5], Vandierendonck and De Bosschere use an SA-110 ARM processor model configured with a 4 KB L1 direct mapped cache and benchmarks from the PowerStone, MediaBench and MiBench suites for experimentation. Particularly interesting are the presented cross profiling results, which demonstrate the generalization behavior of the XOR mapping functions trained on a specific application and input data for different test input data. Two metrics are reported, the reduction of the miss rate and the reduction of the overall run time. While in nearly all cases the miss rates were reduced over a modulo cache, the corresponding run times do not strictly follow this trend and, occasionally, slowdowns were observed.

Our work shares with related approaches the main concept of modifying the mapping between address and cache line index in dependence of the application. We strongly differ, however, in the following novelties: The address mappings for the caches are implemented by more versatile reconfigurable logic circuits and are determined by an evolutionary search technique. Our cache hierarchies are more realistic with split L1 and unified L2 caches, as well as external main memory. The evolutionary optimization is driven by an application's overall execution time which is simulated in a cycle-accurate manner.

The remainder of this paper is organized as follows. Section II introduces the concept of an evolvable cache (EvoCache) and describes its integration into the processor architecture and the operating system. In Section III, we present the models and algorithms used for evolving caches. In Section IV, we evaluate the proposed EvoCache by comparing the results with conventional caches. Finally, we conclude and present an outlook to future work in Section V.

II. The EvoCache Concept

The key idea of the EvoCache approach is presented in Figure 1. 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]–[3]. While Figure 1 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 [4] and XOR functions [5], [6], 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 the paper.

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 this paper, we 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.

III. Representation Model and Optimization Algorithm

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

A. Cartesian Genetic Programming

The evolved logic function consists basically of a set of combinational logic nodes arranged in a two-dimensional grid and connected by feed-forward wires. Additionally, the circuit comprises a number of primary inputs and primary outputs. The grid structure of the circuit is inspired by field programmable gate array (FPGA) architectures and also depicted in Figure 1. Such a hardware representation model is known as Cartesian Genetic Programming (CGP) model and widely used in the evolvable hardware community [7].

Formally, a CGP model is a $(n_r \times n_c)$-grid of nodes. A node can have n_n inputs which connect to global inputs and to nodes in the previous l columns. In the experiments presented in this paper we have used look-up tables (LUTs) with four inputs ($n_n = 4$) 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 one-row CGP model is sketched in Figure 2. The mapping from a circuit evolved in this one-row model to a two-dimensional grid as shown in Figure 1 is straightforward as long as the maximal circuit depth is limited to the number of columns of the grid. 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 in

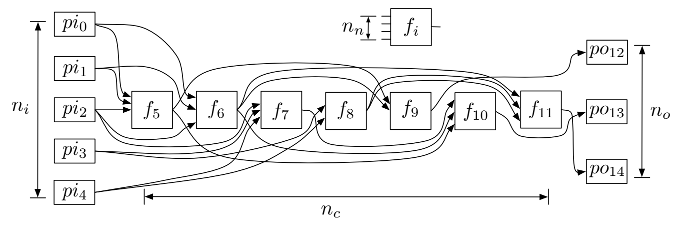


Figure 2. Cartesian Genetic Programming (CGP) model in a one-row configuration.

this paper have been conducted with unconstrained circuit depth. Instead, in Section IV we report on the depths and sizes of the evolved circuits.

B. Evolutionary Strategies

As optimization technique we use an evolutionary strategy (ES). We have also conducted studies with genetic algorithms, even with multi-objective genetic algorithms [8] that simultaneously optimize for run-time, energy, and the delay and area of the reconfigurable logic. Among these algorithms, the ES excels in optimizing for run-time. The results presented in this paper have been achieved with a standard $1 + 4$ 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.

C. Fitness Evaluation and SimpleScalar Integration

For the experiments, we leverage our MOVES EHW toolbox [9], 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 3. 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 [10] 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

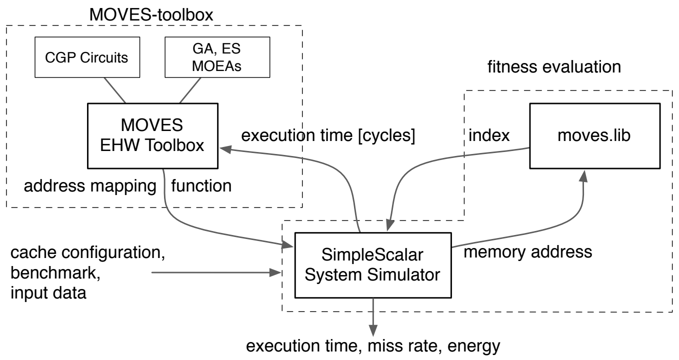


Figure 3. EvoCache tool setup: SimpleScalar is invoked by the MOVES toolbox and returns the overall execution time in clock cycles as a fitness measure.

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.

D. 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 processor 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 [3] 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

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

The actual values in $[nJ]$ for the static energy per cycle and dynamic energy per access are derived from the CACTI cache model [11] 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 [12]. The overall energy for an application run thus adds up to

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

IV. 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 [12]. The configuration is shown in Figure 4 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.

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

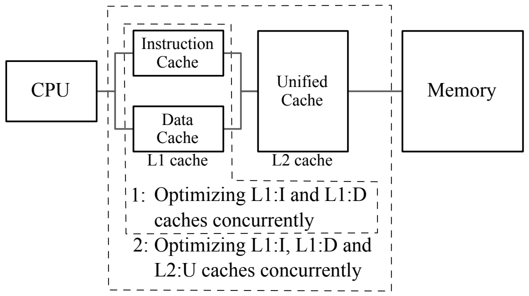

CPU
Instruction Cache
Data Cache
L1 cache
Unified Cache
L2 cache
Memory
1: Optimizing L1:I and L1:D caches concurrently
2: Optimizing L1:I, L1:D and L2:U caches concurrently

Figure 4. 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).

data compressor based on the Burrows-Wheeler transformation [13] and was reported to cause a large amount of cache misses. The picture encoding application JPEG [14] is a commonly used benchmark for performance analysis.

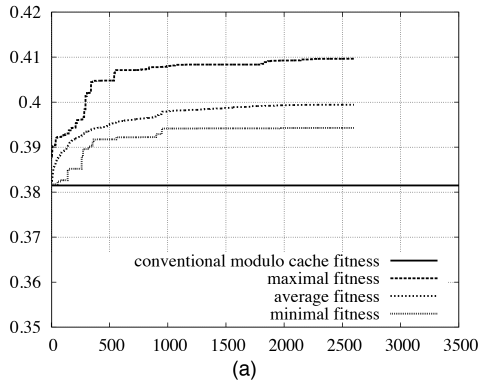
For each combination of benchmark and optimization scenario, we have proceeded as follows. First, we have evolved a mapping function for a given input data set, denoted as training data. This optimization step has been repeated for 16 times. To study the potential of EvoCaches, we analyze the fitness 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. While we have used random mapping functions to initialize the evolutionary optimizer for both benchmarks, we have additionally experimented with modulo mappings as initial individuals for BZIP2.

For BZIP2, the training data set consists of the HTML code from Wikipedia's page on 'Genetic Programming' [15]. 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 [16] and [17].

A. Training EvoCaches

Figure 5 presents results for BZIP2 and optimization scenario two, i.e., evolving address mapping functions for L1:I, L1:D and L2:U simultaneously. In Figure 5(a), the evolutionary strategy started with classical modulo mapping functions, whereas Figure 5(b) shows the fitness development for randomly initialized evolutionary search. The four curves in each graph are plotted against the number



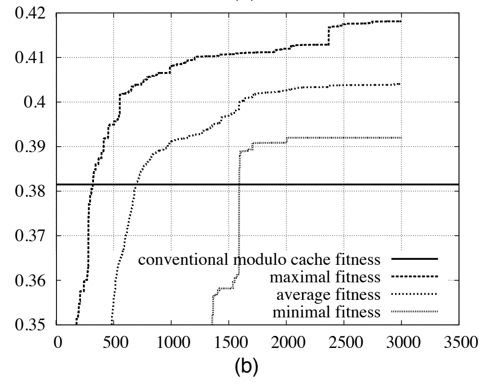


Figure 5. Development of the best, average and worst fitness values over 16 runs for the **BZIP2** benchmark and the simultaneous optimization of split L1 and unified L2 caches. In (a), the evolutionary optimizer has been initialized with the classic modulo mapping function, in (b) a random initialization was chosen.

of generations and picture the fitness which is defined as reciprocal of the overall execution time relative to the cacheless 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 is indicated by a straight line at 0.3815.

The remaining curves in Figure 5 show the maximal, minimal, and averaged fitness values for the evolved mapping functions. The main result is that mapping functions outperforming the modulo cache are easily evolved. Comparing Figures 5(a) and 5(b) we can observe that starting evolution from classical modulo functions is not beneficial. Starting from random mapping functions, we obviously need more generations to beat the modulo cache mapping but the resulting fitness values are overall better and more varied.

Optimization scenario one, i.e., evolving mapping functions for first level caches only, reveals similar behavior. Table I lists the performance gains over the classical modulo cache mappings for both experiments. The table shows the best and average individuals of the 16 runs. The fitness is indicated in columns three and five. For example, the best fitness achieved in the 16 runs for evolving mapping functions for the level one cache (L1:I,D) has been 0.39996 which results in a 4.7% improvement in execution time over the classical modulo mapping function.

optimization scenario		modulo initialization	relative to mod cache	random initialization	relative to mod cache
L1:I,D	avg	0.3952	3.6%	0.4038	5.8%
	max	0.3996	4.7%	0.4086	7.1%
L1:I,D,	avg	0.3994	4.6%	0.4037	5.8%
L2:U	max	0.4096	7.3%	0.4174	9.4%

Table I
PERFORMANCE OF THE AVERAGE AND THE BEST INDIVIDUALS ON TRAINING DATA FOR **BZIP2** AFTER 2500 GENERATIONS.

optimization scenario		random initialization	relative to mod cache
D1:I,D	avg	0.6623	11.2%
	max	0.6975	17.1%
D1:I,D,	avg	0.6718	12.8%
D2:U	max	0.6962	16.9%

Table II
PERFORMANCE OF THE AVERAGE AND BEST INDIVIDUALS ON TRAINING DATA FOR **JPEG** AFTER 1400 GENERATIONS.

Table II summarizes the training results for the JPEG benchmark. Here, we report only experiments with randomly initialized evolutionary searches. 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.

B. 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. For BZIP2, we have selected the four best individuals from the training phase according to the two optimization scenarios and ways of initializing the evolutionary search. 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 III. In this table, the optimization scenario L1-MI denotes optimization of level one caches with modulo initialization, L12-RI the optimization of both levels of caches with random initialization, etc. The numbers for a single benchmark, optimization scenario and initialization technique 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.

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

		BZIP2: ELF benchmark				BZIP2: HTML benchmark				BZIP2: TXT benchmark			
		L1-MI	L1-RI	L12-MI	L12-RI	L1-MI	L1-RI	L12-MI	L12-RI	L1-MI	L1-RI	L12-MI	L12-RI
execution time	best	1.40%	4.92%	3.94%	5.90%	4.00%	5.31%	4.18%	6.98%	5.22%	7.30%	7.03%	10.98%
	average	0.74%	4.36%	3.38%	5.60%	2.74%	3.86%	3.32%	4.94%	2.34%	4.49%	4.08%	6.66%
	worst	0.14%	3.36%	2.36%	4.87%	1.75%	2.47%	2.06%	3.46%	-5.72%	-4.15%	-5.56%	1.95%
miss rate	best	-0.01%	6.11%	5.55%	9.00%	3.76%	5.94%	5.24%	8.92%	4.27%	8.45%	7.27%	11.38%
	average	-0.49%	5.59%	5.02%	8.51%	2.55%	4.15%	3.55%	6.41%	1.48%	4.82%	4.20%	8.26%
	worst	-1.40%	4.13%	3.52%	7.94%	1.08%	1.64%	1.92%	4.88%	-11.00%	-9.31%	-11.18%	2.63%
energy requirement	best	1.64%	4.64%	3.55%	5.49%	4.58%	5.61%	5.09%	7.31%	4.71%	6.88%	6.07%	10.70%
	average	1.08%	4.13%	3.06%	5.23%	3.55%	4.53%	3.87%	5.29%	3.06%	4.93%	4.33%	6.98%
	worst	0.59%	3.27%	2.17%	4.57%	2.34%	3.21%	2.60%	3.77%	-4.53%	-2.83%	-4.33%	2.16%

Table III
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.

- 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.
- The advantage of cache mapping functions evolved from random mapping functions over mappings evolved from modulo functions can be also observed when evaluating the cache with test data and is even more pronounced as in the training phase.

For testing EvoCaches on JPEG, we have selected ten images from [16] and [17]. The detailed results are shown in Table IV 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.

The results of our experiments are summarized in Figure 6. 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.

		L1-RI	L12-RI
execution time	best	14.31%	12.96%
	average	12.73%	10.78%
	worst	11.48%	9.12%
miss rate	best	41.25%	40.35%
	average	37.40%	37.19%
	worst	31.64%	30.46%
energy requirement	best	16.43%	14.46%
	average	14.19%	11.93%
	worst	12.53%	10.49%

Table IV
EVOCACHE GENERALIZATION PERFORMANCE FOR THE JPEG ENCODER TRAINED ON THE SAMPLE IMAGE FROM THE JPEG6A SOURCE CODE DISTRIBUTION.

		BZIP2				JPEG	
		modulo initialized		random initialized		random initialized	
		delay	size	delay	size	delay	size
L1:I,D	avg	4.10	13.50	4.63	16.94	4.38	15.13
	max	6	19	7	22	7	22
	best	5	18	4	18	6	19
L1:I,D L2:U	avg	4.06	14.00	4.19	15.81	4.44	15.44
	max	6	20	8	24	6	22
	best	5	16	3	14	4	17

Table V
AREA AND DELAY PARAMETERS FOR THE EVOLVED RECONFIGURABLE ADDRESS MAPPING FUNCTIONS.

The area (number of 4-LUTs) and the delay (depth of the circuit) parameters for the resulting reconfigurable logic circuits are presented in Table V. 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.

V. CONCLUSION AND OUTLOOK

In this paper, we have presented EvoCaches that rely on two main ideas. First, the function mapping an address

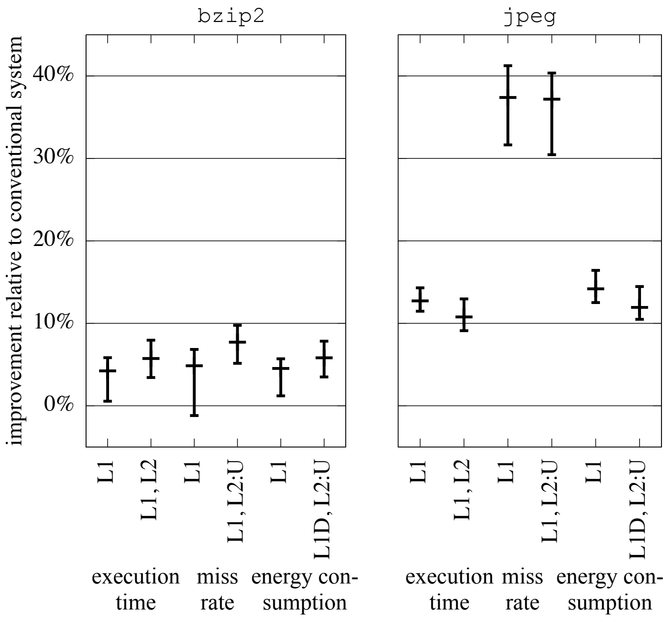


Figure 6. Summary of the EvoCache generalization performance for BZIP2 and JPEG. The data is for randomly initialized mapping functions. The best, worst and average values are indicated for every optimization scenario and metric.

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.

Future work will include the simulation of more benchmarks and cache configurations. To this end, it will be desirable to considerably speed up the fitness evaluation as we are currently limited by simulation time. Therefore we intend to look deeper into the correlation between run-times and miss rates with the goal to use the miss rate as performance metric and thus avoid cycle-accurate simulation. As an alternative, we will investigate whether trace-based simulation can be used to derive reliable results.

Acknowledgment

This work was supported by the German Research Foundation under project number PL 471/1-2 within the priority program *Organic Computing*.

References

[1] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proc. ACM/IEEE Int. Symp. on Microarchitecture*. IEEE CS, 1999, pp. 248–259.

[2] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," *Proc. Int. Symp. on Computer Architecture (ISCA)*, vol. 28, no. 2, pp. 214–224, 2000.

[3] C. Zhang, F. Vahid, and R. Lysecky, "A Self-tuning Cache Architecture for Embedded Systems," in *Trans. on Embedded Computing Systems*, vol. 3, no. 2. ACM, 2004, pp. 407–425.

[4] 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)*. Springer, 2000, pp. 965–968.

[5] 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)*. Springer, pp. 261–272.

[6] H. Vandierendonck, P. Manet, and J. Legat, "Application-specific reconfigurable XOR-indexing to eliminate cache conflict," in *Proc. Design, Automation and Test in Europe (DATE)*, 2006, pp. 357–362.

[7] J. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. Europ. Conf. on Genetic Programming (EuroGP)*. Springer, 2000, pp. 121–132.

[8] T. Knieper, B. Defo, P. Kaufmann, and M. Platzner, "On robust evolution of digital hardware," in *Proc. IFIP Conference on Biologically Inspired Collaborative Computing (BIC)*. Springer, 2008.

[9] P. Kaufmann and M. Platzner, "MOVES: A modular framework for hardware evolution," in *Proc. NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2007, pp. 447–454.

[10] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," vol. 35, no. 2. IEEE CS, 2002, pp. 59–67.

[11] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," COMPAQ Western Research Lab, Palo Alto, California 94301 USA, Tech. Rep., 1999.

[12] "ARM10E processor family," http://www.arm.com/products/CPUs/families/ARM10EFamily.html. [Online]. Available: www.arm.com/products/CPUs/families/ARM10EFamily.html

[13] J. Seward, "bzip2: A Freely Available, Patent Free, High-quality Data Compressor," http://www.bzip.org, 2009. [Online]. Available: www.bzip.org/

[14] G. K. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[15] "Genetic programming," http://en.wikipedia.org/wiki/Genetic_programming. [Online]. Available: en.wikipedia.org/wiki/Genetic_programming

[16] "The USC-SIPI image database," http://sipi.usc.edu/database. [Online]. Available: sipi.usc.edu/database

[17] "Classis test still images," http://hlevkin.com. [Online]. Available: hlevkin.com