Evaluation Methodology for Complex Non-deterministic Functions: A Case Study in Metaheuristic Optimization of Caches

Paul Kaufmann, Nam Ho, Marco Platzner
University of Paderborn
paul.kaufmann@gmail.com

***Abstract*—When evolving a non-deterministic function by Evolutionary Algorithms, a candidate solution is usually evaluated multiple times to estimate its characteristic behavior. This is a valid methodology unless the evaluation is too complex and the fitness evaluations result in unacceptably long optimization times. This challenge can be addressed by either resorting to a simpler surrogate performance model or, in case a surrogate model is not precise enough, by parallelizing search, or by minimizing the number of fitness evaluations if the impact on the quality of search is acceptable.**

The work presented in this paper is motivated by the optimization of processor caches, for which performance evaluation is highly complex and nondeterministic due to the non-deterministic behavior of today's operating systems. Since parallelizing fitness evaluations results in unacceptably prolonged computation times, we employ statistical methods to identify best-performing candidates using as few fitness evaluations as possible. We describe different approaches we have investigated until finally selecting the Wilcoxon rank-sum to adaptively control a fitness evaluation scheme. With this novel scheme we are able to reduce the optimization times by a factor of 3.6 without significant drop in convergence behavior.

I. Introduction

Whenever the size and complexity of a design challenge becomes too large to be solved directly, Evolutionary Algorithms (EAs) offer a viable alternative. Relying on a formal encoding and a set of quality metrics, EAs can solve bigger tasks, although without usually providing any guarantees on the solution quality and the computational time. The evaluation of a quality metric, or fitness evaluation, is usually the most computationally complex task of an EA. In case the behavior of a candidate solution is subject to randomized fluctuations, the fitness evaluation procedure becomes even more complex, as the average behavior has to be estimated in multiple experiments. To reduce the computational complexity of a fitness evaluation, surrogate fitness models are a good choice for design challenges with continuous parameters [1]. Using parametric (e.g. polynomial regression) and non-parametric models (e.g. neural networks), a surrogate fitness landscape can be learned by sampling the original solution space multiple times. After validating the accuracy of the surrogate model, the optimization algorithm can proceed using the computationally inexpensive model. Combinatorial goal functions are more difficult to capture by a surrogate function. Parallelization is therefore a more common approach to minimize the computational time of a fitness evaluation [2].

The motivation of our work on complexity reduction for accurate evaluation of non-deterministic functions is the optimization of processor caches. We use EAs to find Boolean circuits that are mapping memory addresses to cache indexes such that the number of cache misses is minimized. As modern operation systems randomize the dynamic and virtual memory management systems, accurate performance estimation requires multiple runs of an application. Parallelization has not reduced computation times sufficiently.

We have therefore developed a novel adaptive evaluation scheme monitoring the performances of a sequence of application executions and using the Wilcoxon test for stopping the application executions as soon as a statistically sound decision is possible, i.e. difference of mean ranks of two populations can be observed. With this "early-stop" strategy we are able to reduce the computation times by a factor of 3.6 while achieving similar convergence behaviors.

The paper continues with a section on related approaches for optimization of caches. In the third section we present our approach to cache optimization and show the developed architecture. Section four introduces the setup of the optimization procedure and Section five describes the sources of non-determinism in modern processor systems. Section six presents different methods we have tried to reduce the number of program executions to estimate a pairwise ranking of candidate solutions and shows also our final methodology. Then, Section seven summarizes the results and concludes the paper.

II. Related work on Optimization of Caches

Cache optimization can roughly be split into structural and behavior approaches. The goal of structural optimization is to find a good configuration for the underlying cache architecture, e.g. the number of cache blocks, associativity, cache size, and other parameters. When done at design time, structural cache optimization is very well investigated and has a long research history [3]. Reconfigurable hardware technologies allowed shifting structural cache optimization into run-time. One of the first efforts was conducted by Albonesi [4]. The author allowed turning off certain cache ways at run-time in order to save energy while achieving comparable miss rates.

As an extension to this, caches with run-time configurable associativity, replacement policy, and block sizes have been proposed and dynamically tuned for energy in [5] and [6]. A technique for partitioning the Last Level Cache (LLC) has been shown very efficient in [7] and became standard for server processors [8].

Behavioral cache optimization covers the optimization of non-architectural properties of a cache, such as replacement policy and the memory-to-cache address mapping function. Cache block replacement policy remained an acute research area for decades. In contrast, optimization of memory-to-cache address functions has been introduced to Translation Lookaside Buffers (TLBs) of mainframe processors [3] and is since then subject to recurrent research efforts. Before introducing related work on optimization of cache mappings we shortly describe the address translation of a conventional cache.

A. Conventional Memory-to-Cache Address Mapping

A conventional cache structure has a data array memory where each of the 2^m addressable cache lines/blocks contains 2^k words. A modulo-based indexing scheme partitions a memory address $a = (a_{n-1}, \ldots, a_0)$ into the tag, set index, and block offset as can be seen at the top of Fig. 1. With this, the modulo mapping function is computed as $c = [a_{m+k-1} ... a_k]$, picking up m bits of the index segment, and the tag is computed as $t = [a_{n-1} ... a_{m+k}]$. The modulo mapping function is being used in contemporary cache structures due to its hardware design simplicity and good performance for sequences of consecutive addresses.

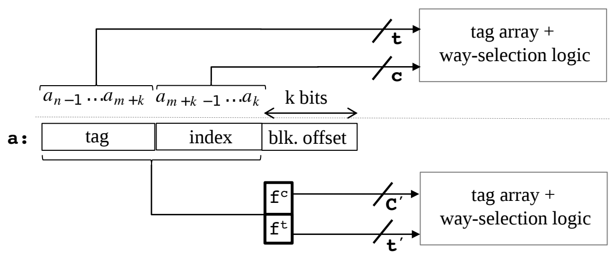

Fig. 1: Simplified cache organizations with modulo/non-modulo indexing schemes.

B. Non-modulo Cache Mapping Functions

A cache working with non-modulo-based mapping functions is shown as a simplified scheme at the bottom of Fig. 1. Here, the functions f^c, f^t are forming additional hardware circuits taking the address $a = [a_{n-1} ... a_k]$ without the block offset bits and computing index and tag bits c' and t'. Optimizing f^c is an alternative way to improve the performances of caches.

In related work, closest to the conventional cache mapping is the permutation-based mapping where f^c is constructed by permuting bits belonging to the index segment of the address a. This approach has been demonstrated in [9], [10]. Similar work has been presented in [11], but with bits selected also from the tag segment of a at run-time. Inspired by the XOR-based TLB hash functions, Vandierendonck et al. introduced a layer of XOR gates for computing the cache index bits and presented a heuristic to find the optimal XOR wiring for some applications [12].

More complex cache address translation functions have been investigated in [2] and [13]. There, the authors have used reconfigurable fabrics of LUTs to compute Boolean circuits of certain size. LUTs' configurations have been evolved by an Evolutionary Algorithm (EA). Leveraging this work, we have developed a hardware implementation of a multi-core system running a full-fledged OS. For this, we have extended the caches and their snooping mechanism of a Gaisler LEON3 SPARC multi-core architecture, realized universal cache and hardware event sensors and incorporated them into the standard Linux performance measurement infrastructure and extended the Linux kernel to handle cache mapping reconfigurations during task switches.

III. Evolvable Cache Architecture

This section covers our multi-core processor implementation able to dynamically evolve and operate reconfigurable cache mapping functions at the first level of caches. We first detail the hardware architecture of reconfigurable cache mappings and their integration in a multi-core CPU architecture. Then, we present the hardware model for the reconfigurable cache mappings as well as the corresponding hardware realization. Finally, we give details on the FPGA prototype.

A. Cache Organization

We focus on Physically Tagged Physically Indexed (PTPI) caches, where the TLB is placed prior to the cache controller. While this decision was motivated by a simpler implementation of a coherent memory model, future work will also investigate virtually addressed caches. As can be seen in Fig. 2, on the core's side the extensions to the conventional cache architecture are the reconfiguration controller (RC) and reconfigurable circuit blocks. The reconfigurable circuit blocks are able to compute any Boolean function of up to a certain size on the address bits $a = [a_{n-1} ... a_k]$ and provide the outputs for indexing cache lines. The configuration bitstreams for the blocks are stored in DRAM. The RC controls the reconfiguration process; the bitstream transfer is done via DMA. The reason for having multiple reconfigurable blocks is to mask the reconfiguration time. The number of blocks can even be increased to reduce the reconfiguration time in massive multi-threaded systems. It is important to note that the tag for non-modulo mapping functions consist of all non-block-offset bits of $a = [a_{n-1} ... a_k]$, which increases the overall size of cache memory.

B. Implementation of a Coherent Memory Model

In physically tagged caches changing the memory-to-cache address translation requires the cache to be flushed. While this increases compulsory misses and introduces overhead, we restrict these effects by scheduling a task always to run on the same core. Therefore, context switches happen infrequently.

In systems with multiple caches that use the snooping protocol to implement a coherent memory model, each time

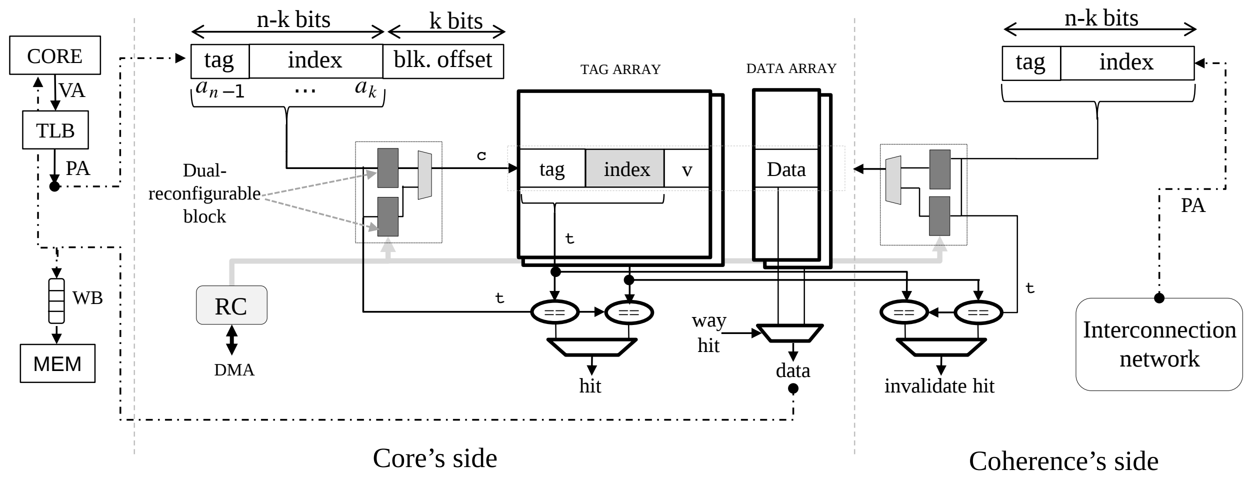

Fig. 2: Level 1 Cache with Reconfigurable Mappings

a core invalidates a cache line, all other cores have to check whether their caches contain a cache line with the same address and invalidate it. As the caches use different cache indexing functions, the snooping mechanisms of each core need to compute the mapping function currently used by the core. Therefore, as shown on the coherence's side of Fig. 2, the snooping mechanisms have also reconfigurable blocks computing the same mapping function as currently used by the core side.

C. *Incorporating Reconfigurable Cache Mappings*

Similar to previous work [2], [13], we encode address translation functions as a Cartesian Genetic Program: a two-dimensional array of run-time reconfigurable 2-input LUTs connected by feed-forward wires. The routing between the combinational nodes is in our case a fixed butterfly network. To give the optimization algorithm more freedom for routing, the first column may connect to any of the address bits. The reconfiguration time for a node is four clock cycles. The RC can configure up to 32 nodes concurrently. For example, reconfiguration for a grid of 80 nodes, organized as 16 rows $\times$ 5 columns, takes 12 clock cycles.

D. *Putting it All Together and Prototyping on an FPGA*

Fig. 3 shows an overview of our prototype on a Virtex-6 FPGA. We implement the Cartesian Genetic Programming (CGP) grid by mapping the combinational nodes of the model to Xilinx's SRLC32E primitives, one by one. These primitives use native look-up tables (LUT) of the FPGA and have a similar structure as CGP nodes described above. However, since these primitives on a Virtex 6 FPGA have 5 inputs, we use only the first two for implementing a CGP node [14]. We plan to use the full width of the native Xilinx's LUTs in future work.

The RC is implemented as an additional hardware module, working in cooperation with a DMA interface. It includes control registers that can be accessed by any core via a dedicated bus interface. To this end, we have extended the Address Space Identifier (ASI) lda/sta instructions of the SPARC architecture in LEON3 [15].

Lastly, we have implemented a measurement infrastructure allowing us to monitor the performance of the caches and the according address mapping functions. The infrastructure incorporates into the standard Linux kernel monitoring tool for collecting microarchitectural metrics [16].

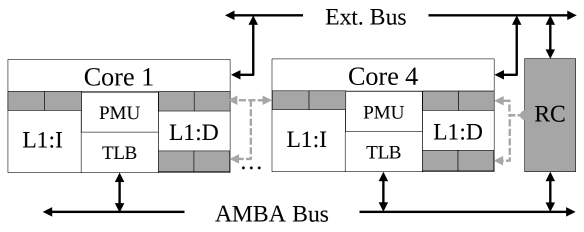

Fig. 3: Implementation of reconfigurable cache mappings in a multi-core LEON3. Gray blocks are extensions to the regular LEON3 architecture (reconfigurable cache mappings for Integer and Snooping Units and the Reconfiguration Controller).

E. *System Specification*

Table I summarizes parameters of our prototype system supporting reconfigurable cache mappings in a quad-core LEON3 platform. The prototype is implemented on a ML605 board equipped with a Virtex-6 FPGA. We have implemented reconfigurable circuits according to the CGP model for both L1:I and L1:D caches. Using corresponding device drivers, the system can execute Linux and reconfigure the cache mappings at run-time.

Table II shows the hardware resource usage for one core of the quad-core platform synthesized with $4KB, 1-way$, $8KB, 1-way$ and $8KB, 2-way$ memories for instruction and data caches. The RC is shared by all cores and implemented as an independent component with FIFOs for fetching reconfiguration data from memory via DMA. The implementation consumes 13 Distributed RAMs (DRAMs), where $RAM32x1D$ primitives are used. Based on a previous implementation [16], the Performance Monitoring Unit (PMU) per core is extended to support up to 8 concurrently monitored hardware events. The reconfigurable fabrics are implemented with three dual-blocks, out of which one is dedicated for L1:I, and two are for L1:D caches. Each of the CGP grids has 80 nodes, instantiated as 80 SRLC32Es.

TABLE I: LEON3 quad core implementing reconfigurable cache mappings.

Generic System Configuration	
Parameters	Configuration
Clock Frequency	50MHz
Floating Point	Software
Memory	1GB DRAM
I/D-TLB	8 entries
Linux Kernel	2.6.36.4 patch from Gaisler
Compiler	Pre-built Linux toolchain from Gaisler
PMU	8 event counters
RC	Reconfiguration Controller
Cache Configuration	
L1:I & L1:D	4KB:1-way, 8KB:{1,2}-way, {16,32}-bytes/line
Coherency	Snooping Protocol

TABLE II: Hardware resources used by a one-core system. The overhead[%] compares the resources to a regular LEON3 implementation with a conventional cache.

	FFs	**LUTs**	**DRAMs**	**BRAMs** [2]
RC [1]	176	557	13 *(RAM32x1Ds)*	0
PMU	401	1258	0	0
CGP & Controllers	2972	1558	80 x 6 *(SRL16Es)*	0
Cache Controllers				
4KB,1-way	969	2543	0	0
Overhead[%]	39.4%	23.8%	0.0%	0.0%
8KB,1-way	1238	3106	0	0
Overhead[%]	29.6%	14.9%	0.0%	0.0%
8KB,2-way	1246	3288	0	0
Overhead[%]	29.3%	35.5%	0.0%	0.0%
Cache Tags & Memories				
4KB,1-way	46	47	0	7
Overhead[%]	21.1%	17.5%	0.0%	0.0%
8KB,1-way	48	48	0	12
Overhead[%]	23.07%	23.07%	0.0%	9.1%
8KB,2-way	92	90	0	14
Overhead[%]	21%	25%	0.0%	0.0%

IV. The Evolutionary Optimization Procedure

The optimization procedure for cache mappings consists of two phases. In the training phase an (1+4) Evolutionary Strategy is searching for good-performing cache mappings and in the validation phase the best individual found so far is evaluated on data not used in the training and the numbers are reported.

The optimization procedure is sketched in Fig. 4. A candidate solution presenting the configuration of the functional blocks and the routing of primary inputs of the reconfigurable address mapping blocks of LEON3 is encoded by a bitstring. When starting the evolution, the initial population of one candidate solution (parent) is either sampled randomly or it encodes the conventional modulo cache mapping. After initialization, a loop is iterated for a predefined number of cycles (generations) creating in each iteration four offspring individuals (children) by duplicating the parent's bitstring and mutating it. Mutation is defined as randomly flipping few bits. Each child is evaluated with the mean number of misses per kilo instructions (MPKI) for an application and a set of input data vectors. For the evaluation procedure, the reconfigurable memory mapping fabrics are configured by the bitstream of the candidate solution and the target application is executed for each of its training input vectors. Using Linux's `perf_tool` the performance numbers are monitored and reported to the ES algorithm. ES collects the functional qualities and select the best individual as the new parent. The old parent proceeds to the next generation only if it is strictly better than all of its offspring individuals.

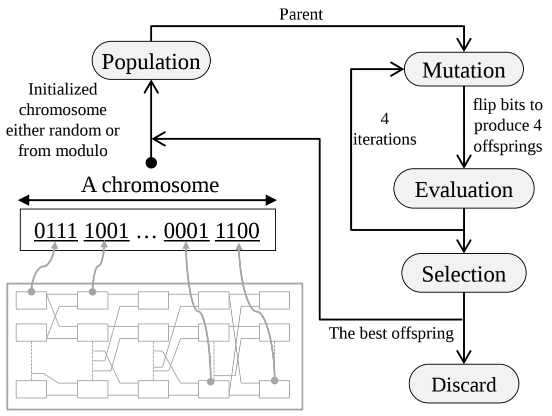

Fig. 4: Evolution of Caches: $(1+4)$-ES.

V. Sources of Non-determinism and the Evaluation Procedure

Realistic estimation of performance of a computing system depends on various direct and implicit factors. Usually, the performance is derived for an application and its input data of certain size. Factors like the interdependence with concurrently executed applications competing for the same resources and the overhead by the performance measurement system are minimized as far as possible. But even then experiments repeated under identical conditions still may produce varying performance numbers. In the following we present two sources of non-determinism making it difficult to compare processors with different cache mappings. This initiated our search for an accurate and efficient performance evaluation scheme.

[1]Shared by all cores
[2]BRAMs, different data widths

A. Randomization of Physical Page Allocation

We evolve cache mapping functions on the basis of memory access patterns of applications. Memory access patterns change depending on various factors. These factors have to be considered to evolve cache mappings with good general performance. The size of the input data, as already mentioned, is the first factor that changes data and instruction memory access patterns. For algorithms with execution order of instructions depending on the values of the processed data, the statistical distribution of data values is the next impact factor. For systems with dynamic memory management and a physical address space, an application is usually loaded by the OS into different address spaces, depending on the currently executed application set. For systems with virtual address spaces the loader usually assigns the same virtual address ranges to program segments. However, the mapping from virtual to physical pages is usually randomized by the OS. As our processor implementation is using physically addressed L1 caches, almost all mentioned factors are relevant for our situation and we observe different memory access patterns for each re-execution of an application. Hence, a characteristic performance of a candidate cache mapping has to be derived in multiple experiments.

B. Deviation of Cache Mapping Functions

A memory-to-cache mapping can be seen as a special form of a hash function. The quality of a hash function f, $f(x \in \mathbb{B}^n) \rightarrow y \in \mathbb{B}^m$, $n > m$, is usually defined as how evenly input values $x \in \mathbb{B}^n$ are distributed among the outputs of $f(x) \in \mathbb{B}^m$. In contrast, cache mapping functions act on unevenly distributed input values where also the temporal occurrence of the values has an impact on f's quality. The consequence is that for good cache performance the output distribution of f may need to be highly irregular. As can be anticipated, irregular hash functions tend to deviate in their outputs for varying input sequences stronger than uniform hashes. An example is given in Fig. 5. There we compare the number of L1:D cache misses in 10×10^6 instruction executions of the CJPEG executable when configured to use the conventional and an optimized cache mappings. The numbers are reported as sample means and deviations after 5, 10, 15…40 iterations. The first observation is that the modulo caching function (white bars) is much more consistent in the number of misses per instruction than an evolved hash function (gray bars). The deviation becomes smaller for both mapping functions when the number of experiments increases.

A surprising observation is that while for 5, 10…35 iterations the evolved non-modulo cache mapping is consistently better, for 40 experiment iterations the modulo-based caching mapping excels. This illustrates the challenge we are facing when comparing different cache mapping functions while simultaneously trying to minimize the computational complexity of the comparison.

C. Functional Quality and the Evaluation Procedure

The objective of the optimization algorithm is to reduce the number of cache misses. We use the Miss Per Kilo Instructions

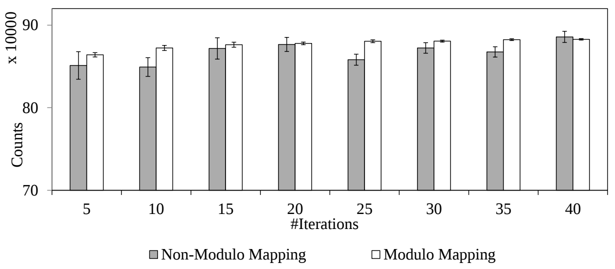

Fig. 5: Sample mean and deviation of L1:D's $misses$ computed by looping CJPEG executions.

(MPKI) as the goal metric defined as

$$\text{MPKI} = \frac{M}{IC} \times 100,$$

where M and IC are the numbers of $misses$ and $retired\ instructions$, respectively.

As the search for good performing cache mapping functions has to ensure that candidate solutions excel for a wide range of potential input vectors, we evaluate candidate solutions on multiple input vectors. The input vectors are selected to be as different and as representative as possible. To be able to aggregate MPKI values for different input vectors, we normalize the values to the performance of the modulo cache mapping function. That is, for an application app, an input_vector $\in I = \{i_1, i_2, i_3, i_4\}$, a candidate cache mapping function candidate and the reference modulo caching mapping function modulo, the **normalized MPKI** is defined as:

$$\texttt{MPKI}_1 = \frac{\texttt{MPKI(app, input\_vector, candidate)}}{\texttt{MPKI(app, input\_vector, modulo)}}.$$

$\texttt{DMPKI}_1$ and $\texttt{IMPKI}_1$ represent the $\texttt{MPKI}_1$ metric for the data and instruction caches of an split L1 cache, respectively. Sequences of normalized $\texttt{DMPKI}_1$ values are the basis for mutual comparisons of candidate solutions used by the evolutionary strategies. Before detailing the comparison procedure in the next section, we would like to present the MPKI evaluation scheme. Our LEON3 implementation includes four cores. When profiling an application for MPKI metric, we parallelize up to four executions while forcing the execution of a candidate application to a dedicated core with Linux's affinity feature to minimize interdependencies. A restriction we made is that the candidate application, its input vector, and the cache mapping function have to be the same for all cores. This restriction is, however, motivated by the simplicity of the evaluation procedure only. In regular operation mode all LEON3 cores may select their own hash mapping function that also can be reconfigured during any task switch.

VI. Search for an Efficient Evaluation Procedure for Non-deterministic Objective Functions

In this section we present insights taken from the evolution of cache mappings for the CJPEG application. The same observations are valid for other benchmarks from the `MiBench` suite, like `BZIP2`.

A. Reference Performance

Before investigating ideas for the reduction of computational effort for fitness evaluation of non-deterministic functions, the baseline performance for an exemplary benchmark is established and described in this section. We have selected the `CJPEG` benchmark from the `MiBench` suite and evolved in three experiments by a $(1+4)$ ES executed for up to 2000 generations cache mappings for a direct mapped 4kB L1:D cache with cache blocks of four 4-byte words. As described in the previous section, fitness evaluation of a candidate cache mapping was conducted for four input vectors consisting of 256 by 256 pixels images. All three experiments were started from randomly initialized solutions. The number of iterations an application and the according cache mapping function were executed was set to $K = 32$.

Dashed lines in Figure 6 show the development of $\texttt{DMPKI}_1$ evolved in three runs by $(1+4)$ ES for $K = 32$. The thick solid horizontal line indicates the baseline performance of the modulo cache mapping. The first observation is that all runs are able to evolve a cache mapping that is at least as good as the conventional cache mapping after 2000 generations. The second and third runs are able to reach the break-even after roughly 250 generations minimizing the cache misses of the L1:D cache by 20% and more than 30%, respectively. While the results are promising, the execution time for a single run amounted roughly for 12 days. Using three Xilinx ML605 boards we were able to finish all three runs in the mentioned time. However, when the goal is to evolve cache mapping functions for all applications of a benchmark suite, like SPEC and MiBench, and especially use larger input vectors, the trend of the observed computational complexity becomes prohibitive.

B. The Racing Procedure

To reduce K we propose an adaptive scheme executing two algorithms for few times and using the Wilcoxon test to identify whether the medians are significantly different. In case of a tie, the algorithms are executed for an additional round and the test is repeated. This procedure continues until a winner has been found or the computational budget expires. In the latter case the population with the better median wins.

Normally or not Normally Distributed: That is the question!

When comparing two populations whether one has a better mean, a common method is to compute the p-value for the distribution of the distance between sample averages. This, however, requires that the populations follow the normal distribution. Using the Shapiro-Wilk, Kolmogorov-Smirnov, and Anderson-Darling tests for $\alpha = 5\%$ we have investigated the normality of the $\texttt{DMPKI}_1$ sequences with the results, that 47.9%, 41.1%, and 46.7% are not following the normal distribution. This forced us to resort to non-parametric methods.

The Evaluation Procedure

Inspired by the iRace algorithm racing package [17], we have implemented initially the Friedman's test to decide, which of the five candidate solutions of a generation produce a better population of $\texttt{DMPKI}_1$ numbers. As the test relies on paired treatments, which in our case is an unnecessary restriction, we have resorted at the end to the Wilcoxon rank-sum test. With this, we are following the racing procedure described previously with a maximal computational budget of $K = 16$ iterations. The final evaluation scheme is sketched in Algorithm 1.

Algorithm 1: `Adaptive Parallel Evaluation`

```
Input:
– app: candidate application
– $p, F_p$: parent individual and its $K$ $\texttt{DMPKI}_1$ values
– $C = \{c_1, c_2 \dots\}$: a population of candidate solutions
– $i \in I = \{i_1, i_2 \dots\}$: input vectors
– $k = 4$: $k$-core LEON3 processor
– $K = k * j,\ j \in \mathbb{N}_{>0}$: number of iterations for an input vector $i$
– $\alpha \leftarrow 0.05$: significance level for the Wilcoxon test
Output:
– (best, $F_{\text{best}}$): best individual and according $\texttt{DMPKI}_1$ values
1  $F = \{F_{c_1}, F_{c_2} \dots\} \leftarrow \emptyset$
2  $C' \leftarrow C$
3  forall $c \in C$ do
4      for $l \leftarrow 4, 8 \dots K$ do
5          forall $i \in I$ do
6              for $p = 1 \dots k$ do in parallel
7                  $F_c \leftarrow F_c \cup \texttt{CORE}_p(\texttt{DMPKI}_1(\texttt{app}, i, c))$
8          if wilcoxon$(F_c, F_p, greater) < \alpha$ then
9              $C' \leftarrow C' \setminus c$
10             break
11 $C' \leftarrow C' \cup \{p\}$
12 while $||C'|| > 1$ do
13     pick a pair $a, b \in C'$
14     if wilcoxon$(F_a, F_b, greater) < \alpha$ then
15         $C' \leftarrow C' \setminus c_a$
16     else if wilcoxon$(F_b, F_a, greater) < \alpha$ then
17         $C' \leftarrow C' \setminus c_b$
18     else if median$(a) \geq$ median$(b)$ then
19         $C' \leftarrow C' \setminus c_a$
20 return $(c \in C',\ F_c)$
```

The idea of the algorithm is to use the $k = 4$ parallel cores of the LEON3 processor to evaluate $k = 4$ times an application and its candidate cache solution c on the same input vector $i \in I$. This is repeated for all input vectors $i \in I$ and then the Wilcoxon test is used to drop early those individuals that are worse than the parent individual p (cf. lines 8-10 in Algorithm 1). If after evaluating the candidate cache c four times on all input vectors the Wilcoxon test does not indicate that c is inferior to parent p, a next round of four evaluations for all input vectors is started to increase the population of

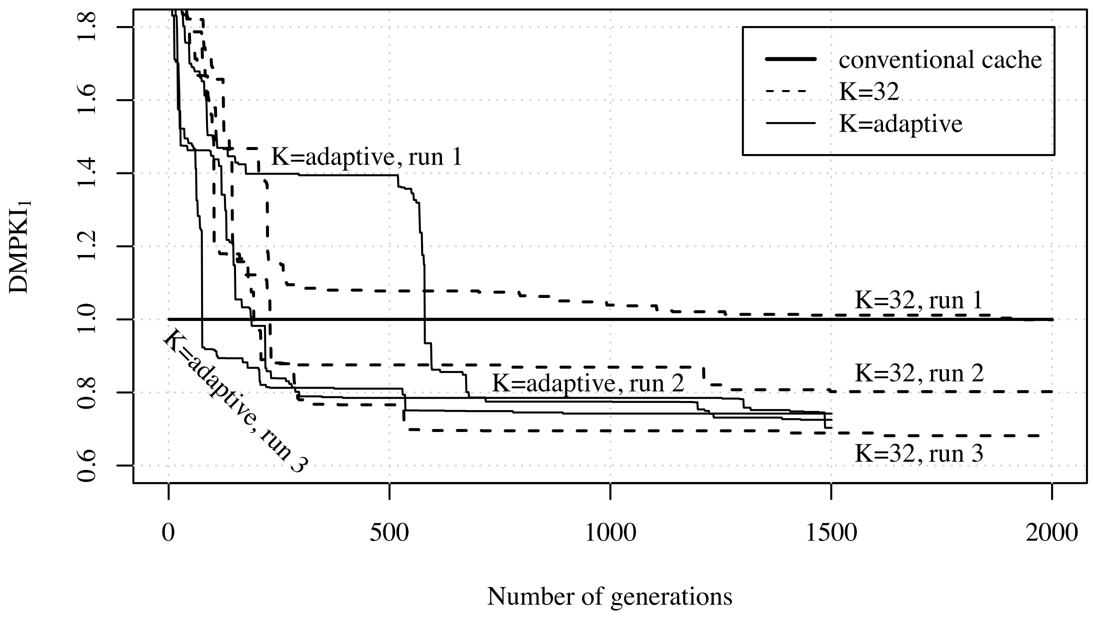

Fig. 6: Evolution of $\texttt{DMPKI}_1$ (performance of best candidate solution) for the L1:D cache by $(1+4)$ ES for $\texttt{CJPEG}$. All experiments have been started from a randomly initialized candidate solution. Dashed lines present the results of three ES runs where the number of iterations K for an input vector was fixed to 32. Solid lines present the results of three ES runs where K was set adaptively, depending on the outcome of the Wilcoxon test comparing five individuals of a generation. The thick solid horizontal line indicates the baseline performance of the modulo cache mapping.

$\texttt{DMPKI}_1$ values of c. The Wilcoxon test is repeated again for the larger sample population of c and if early exit is still not possible, the whole procedure is repeated until the maximal number of fitness evaluations K per input vector $i \in I$ has been reached.

After the first part of the algorithm (lines 1–10), all remaining candidate solutions stored in the set C', and the parent individual have been evaluated K times on each input vector. The remaining candidate solutions are also not worse than the parent individual regarding the Wilcoxon test. In the second part of the algorithm (lines 11–19) all individuals of $C' \cup \{p\}$ that are worse than any other individual in the same set under the Wilcoxon test are removed (lines 14–17). For the remaining individuals the individual with the lowest median $\texttt{DMPKI}_1$ value is selected as the new parent for the next generation.

Reduction of Computational Complexity

The convergence behavior with the adaptive parallel evaluation procedure is compared to the reference behavior in Fig. 6. We have executed the adaptive scheme for 1500 generations. The first observation is that the adaptive scheme still show a convergent behavior and that the deviation of $\texttt{DMPKI}_1$ values is smaller that for the reference case. The quality of the results is roughly on par with the reference evaluation procedure. In Tab. III the distributions of number of (application, input vector) executions in a generation for the three runs is presented. There it can be seen that a decision, which individual is best in a population, can be carried out in 84.83% of the cases after generating only eight $\texttt{DMPKI}_1$ values for each of the four individuals, assuming all four new candidates stay in the race until a final decision is done. Overall, the adaptive scheme allows us to finish an evolutionary run in 3.5 days or 299840 seconds, which is a 3.6 times reduction of the computational time. As in our case the overall optimization time amounts for months, this is a significant improvement. The reduction is not a high as 74% (cf. Tab. III) due to the constant overheads of the performance measurement subsystem.

$l\|I\|k$	1st run	2nd run	3rd run	average	$\frac{K_{\text{adap}}}{K_{\text{ref}}}$
64	0.07%	0.07%	0.07%	0.07%	0.01%
128	81.28%	89.34%	83.88%	84.83%	21.21%
160	16.26%	9.19%	14.06%	13.17%	4.12%
192	2.27%	1.13%	1.93%	1.78%	0.67%
224	0.07%	0.27%	0.07%	0.13%	0.06%
256	0.07%	0.00%	0.00%	0.02%	0.01%

TABLE III: Distribution of number of (application, input vector) executions in a generation for $(1+4)$ ES. The last column compares the average number of (application, input vector) executions among three runs to the reference case with $K = 32$ (i.e. $K|I|k = 32 \cdot 4 \cdot 4 = 512$). The overall reduction amounts roughly to $100\% - 21.21\% - 4.12\% - 0.67\% \approx 74\%$.

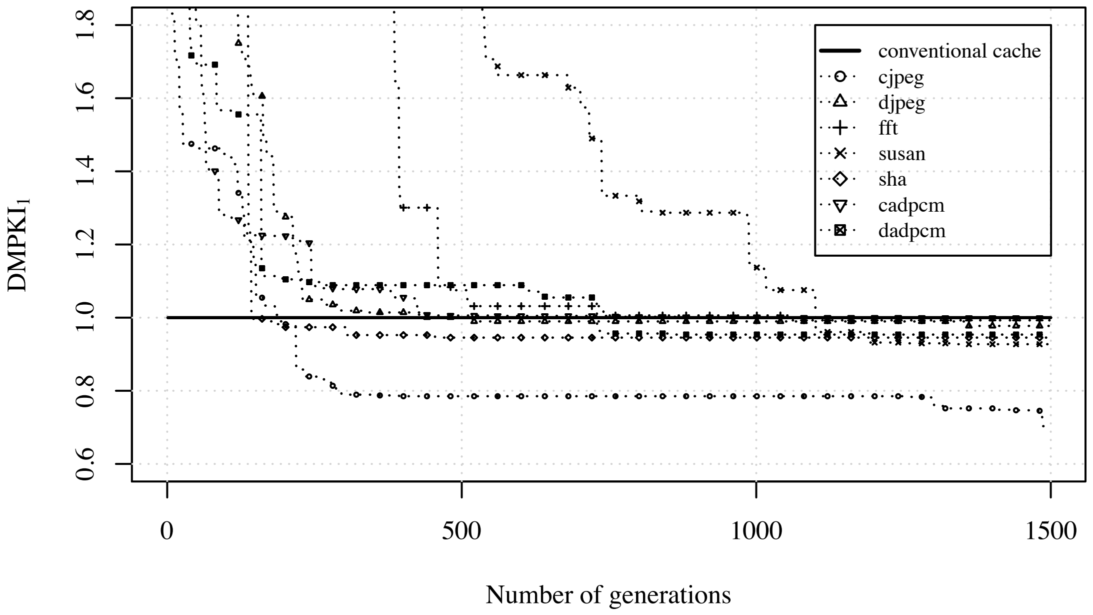

Fig. 7: Training convergence behaviors for some benchmarks from the `MiBench` suite. The experiment configurations are identical to the configuration of the `CJPEG` benchmark.

VII. Conclusion

In this paper we have described the creation of an adaptive parallel fitness evaluation scheme for non-deterministic goal functions. We have shown that we can reduce the computational times by a factor of 3.6 while achieving similar convergence behaviors. Apart from further improving and fine tuning our evaluation method, our main interest for future investigations is whether combining the $\mathtt{MPKI}_1$ values of different input vectors for statistical testing can be replaced by multivariate methods identifying input vectors and input data distributions that may profit from a dedicated hash mapping function. That is, when the $\mathtt{MPKI}_1$ values for a single or a group of input vectors for some cache mapping function constantly underperform compared to the median performance and compared to other cache mapping functions, a dedicated cache mapping function for the outlier group could be beneficial.

References

[1] Y. V. Pehlivanoglu and B. Yagiz, "Aerodynamic design prediction using surrogate-based modeling in genetic algorithm architecture," *Aerospace Science and Technology*, vol. 23, no. 1, pp. 479 – 491, 2012.

[2] Details omitted due to double-blind reviewing.

[3] A. J. Smith, "Cache Memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[4] D. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proceedings. 32nd Annual International Symposium on Microarchitecture (Micro)*. IEEE, 1999, pp. 248–259.

[5] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," *ACM Trans. Embed. Comput. Syst. (TECS)*, vol. 3, no. 2, pp. 407–425, May 2004.

[6] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. Irwin, and A. Sivasubramaniam, "Leakage energy management in cache hierarchies," in *Proceedings on Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2002, pp. 131–140.

[7] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006, pp. 423–432.

[8] Intel, "Improving Real-Time Performance by Utilizing Cache Allocation Technology," Intel, Tech. Rep., 2015.

[9] T. Givargis, "Improved indexing for cache miss reduction in embedded systems," in *Proceedings Design Automation Conference (DAC)*. IEEE, 2003, pp. 875–880.

[10] K. Patel, E. Macii, L. Benini, and M. Poncino, "Reducing cache misses by application-specific re-configurable indexing," in *Proceedings of the 2004 IEEE/ACM Intl. Conf. on Computer-aided Design (ICCAD)*. IEEE Computer Society, 2004, pp. 125–130.

[11] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García, "Adaptive selection of cache indexing bits for removing conflict misses," *IEEE Trans. Computers*, vol. 64, no. 6, pp. 1534–1547, 2015.

[12] H. Vandierendonck, P. Manet, and J. Legat, "Application-specific reconfigurable xor-indexing to eliminate cache conflict misses," in *Proceedings Design, Automation and Test in Europe (DATE)*. IEEE, 2006, pp. 1–6.

[13] Details omitted due to double-blind reviewing.

[14] Xilinx, "Virtex-6 libraries guide for hdl designs." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/virtex6_hdl.pdf

[15] Aeroflex Gaisler, "Grlib." [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf

[16] Details omitted due to double-blind reviewing.

[17] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43 – 58, 2016.

Algorithm 2: `compare`

```
Input: $p$, $F_p$ - parent candidate solution and its $\texttt{MPKI}_1$ values
Input: $c$ - off-spring candidate solution
Input: app, $I_0 \dots I_3$ - application and its input vectors
Input: $n \leftarrow 12$ - max. repetitions: (application, input vector)
1 $|F_c| \leftarrow \emptyset$
2 for $i \leftarrow 1, \dots, n$ do
3     $F_c \leftarrow F_c \cup \{\texttt{MPKI}_1(\texttt{CPU}_0, \text{app}, c, I_0) \dots \texttt{MPKI}_1(\texttt{CPU}_3, \text{app}, c, I_3)$
4     if Wilcoxon.ranksum($F_c$, "worse", $F_p$, $\alpha \leftarrow 0.05$) then
5         return ($p$, $F_p$)
6 if median($F_p$) $<$ median($F_c$) then
7     return ($p$, $F_p$)
8 return ($c$, $F_c$)
```