Microarchitectural Optimization by Means of Reconfigurable and Evolvable Cache Mappings

authors omitted for blind review

***Abstract*—Physical limits are pushing chip manufacturer towards multi- and many-core architectures to maintain the progress of computing power. This trend has also emphasized reconfigurable computing, which enables for even higher parallelization degrees. Reconfigurable computing is often used together with a conventional processor to accelerate highly specific applications. However, exploiting dynamically reconfigurable systems for microarchitectural optimization is a novel research area. This paper presents for the first time an FPGA-based implementation of a processor that can reconfigure and adapt its own memory-to-cache address mapping function at runtime by means of dynamic reconfiguration and nature-inspired optimization. In experiments we can achieve up to 7.8% better execution times compared to a processor with a conventional cache mapping function.**

I. Introduction

When looking back at the history of computer systems, the performance of central processing units (CPU) has grown twice as fast as the performance of DRAM main memories [1] for a long time, creating a gap between these tightly coupled elements. Exploiting the principles of temporal and spatial locality in instruction and data accesses, computer architects have introduced memory hierarchies placing several levels of rather small SRAM-based memories, so-called caches, between a CPU and its main memory. The closer such a cache is placed to the processor's register in the memory hierarchy, the faster and smaller it will be compared to main memory. In case of a cache hit in the first level of caches, a processor can fetch instruction and load/store data vectors typically without any noticeable delay. However, if the desired memory vector is not in the cache, it has to be fetched from lower level caches or main memory stalling the processor for tens or even hundreds of clock cycles. Thus, efficient caches are fundamental to the performance of modern processors.

Research on improved caches is continuing to date, because caches consume a significant amount of a CPU die contributing not only to the performance of a CPU but also to its costs and power consumption. Architectural aspects such as size, associativity, replacement strategies, and block sizes have been investigated with respect to the total costs, area, power consumption, and performance [2], [3]. Particularly in the area of embedded systems with a small and often known-in-advance set of applications, highly tailored memory and cache systems are very promising. Here, self-tuned reconfigurable caches have been investigated in [4] and [5].

Architectural cache tuning is not the only way of improving, tailoring and adapting a cache. Non-architectural properties such as the replacement strategies and custom memory-to-cache address mapping functions have also received attention. While replacement strategies have been investigated exhaustively, mapping function modifications have been analyzed only by few papers. Conventionally, a modulo function is applied to the main memory address to compute the corresponding number (index) of a cache line within the cache. This scheme is popular since it has no temporal and resource overhead in case the number of cache lines is a power of two. However, one can imagine having multiple memory-to-cache address mapping functions tailored to different applications resulting in better execution times. The limited research on customized memory-to-cache address mapping functions presumably is due to the more complicated simulator set up and prolonged simulation times. Moreover, to the best of our knowledge no such system has been build yet.

The novel contribution of this paper is the first FPGA implementation of a processor architecture with an evolvable cache structure. We detail the architecture which is able to execute programs using custom cache mapping functions and show how to evolve those cache-mapping functions. The resulting prototype allows for improving the computation time and, in perspective, also the energy consumption, and gives the operation system advanced control over the cache including, for instance, cache partitioning.

In the remainder of the paper, we first review related work in Section II and then describe the concept of a reconfigurable cache mapping as well as the integration into the LEON3 platform in Section III. In Section IV we present the bio-inspired optimization algorithm used to evolved the cache mappings. In Section V, we present experimental results using MiBench workloads. Finally, Section VI concludes the paper and outlines future work.

II. Related Work

The conventional mapping of memory addresses to cache lines takes a subset of address bits and uses them for indexing a cache line, typically the least significant k bits of the address besides byte and block offsets. If n is the number of cache lines, $k = \log_2 n$ bits are required to index the cache lines. Recently, the research community started to investigate novel mapping functions. Three approaches to map address to index bits have been evaluated so far: the permutation of index bits [6], XOR functions [7] and arbitrary Boolean circuits [8].

Regarding the first approach, Givargis et al. [6] have presented a heuristic that finds for a specific application a subset and permutation of the memory address bits used for selecting a cache line. The heuristic is guided by the miss rate. While the approach has shown promising miss rate reductions, the question of how to load/replace an application-specific mapping function during a context switch was not discussed.

Vandierendonck et al. take a step further introducing a layer of XOR gates computing the index bits [7]. An algorithm evolves the connection pattern between the memory address bits and inputs of the XOR gates. Using a 4KB cache and the PowerStone benchmarks, the authors have evolved and cross-validated cache mappings and shown that their mappings produce almost always better miss rates than the conventional mapping.

Lately, Kaufmann et al. [9], [10], [11], [12], [13] have introduced another approach of reconfigurable cache mappings, called EvoCaches. While their approach is conceptually similar to the related works, it strongly differs in the way they develop and implement the cache mapping functions. The mapping functions are modeled as generic Boolean circuits that are evolved by a genetic algorithm. For example, in [9] the authors have investigated a fully fledged cache similar to modern ARM architectures with a split L1 cache as well as a uniform L2 cache. The have evolved for each of the caches a separate mapping function and used the total execution time rather than the miss rate to guide the search. The cross validation results have shown improvements of up to 14% in execution time, up to 16% in energy consumption and up to 40% in miss rate reduction.

Focusing on adaptable systems and on the evolution of novel cache mappings, this paper presents an FPGA implementation of a processor with run-time self-reconfigurable cache mappings. In order to realize this idea, we have selected the LEON3 processor [14] instantiated on a Virtex 6 FPGA and implemented all caches as well as a measurement infrastructure allowing us to finely monitor the execution times and cache performances in the reconfigurable fabric.

III. Reconfigurable Cache Mapping

A. The EvoCache Concept

Inspired by earlier work on EvoCache, we take this concept further and present a multi-core architecture with distributed caches that allows us to deploy and evaluate the EvoCache idea directly on a reconfigurable hardware platform. An evolvable cache consists of small reconfigurable fabrics woven into the address paths of caches and an optimization algorithm, searching for good cache mappings and reconfiguring the fabrics. Figure 1 presents our architecture, in which the gray parts denote partial reconfigurable fabrics dedicated for reconfigurable cache mappings. For each CPU we need redundant reconfigurable fabrics that snoop the inter-CPU bus and help detecting write back and write through collisions. In case of virtually addressed caches, our architecture needs to be extended by an additional collision unit. These units are not presented in Figure 1.

We encode candidate solutions for memory-to-cache address mapping functions using the Cartesian Genetic Programming model (CGP) [15]. CGP is well suited to represent combinational logic circuits as it encodes a two dimensional grid of functional nodes connected by feed forward wires. Our CGP implementation is shown on the right-hand side of Figure 2. There are many possibly mappings of CGP encoded circuits to FPGA logic. In this work, we map CGP nodes to native look-up tables (LUT) of an FPGA and fix the routing between the nodes in the CGP model as well as on the FPGA to a butterfly network. To give the optimization algorithm more freedom for routing, LUTs in the first column may connect to

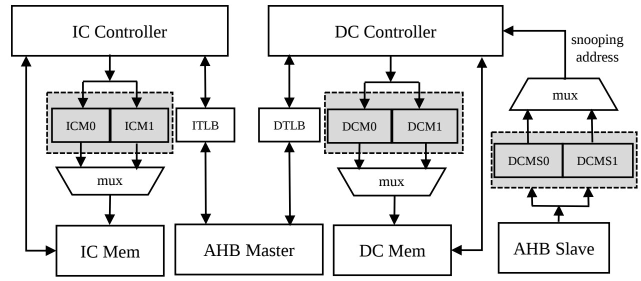


Fig. 1: The system with reconfigurable cache mapping supports. Abbreviations: IC/DC = instruction/data cache; {I/D}M{0/1/S0/S1} = instruction / data / cache mapping function {0/1/snooping 0/snooping 1}

any of the address bits. The input routing configuration is also part of the evolutionary search process. The final architecture is therefore quickly reconfigurable, as only few FPGA LUT contents need to be changed.

In order to make EvoCache work with our architecture, we have allocated two partially reconfigurable functions for each cache mapping. As showed also in Figure 1, instruction cache structure has additional two mappings, named ICM0/1 (Instruction Cache Mapping). Similarly, data cache has two mappings, DCM0/1 (Data Cache Mapping). This way, while one mapping is operational the other mapping can be reconfigured without interfering with the system. Once reconfiguration of a mapping is done, the system has to flush the cache before switching to the new mapping. As we are targeting multi-core systems and the LEON3 architecture realizes cache coherence by a snooping protocol, we have also introduced reconfigurable functions, DCMS0/1 (Data Cache Mapping for snooping operation) shown in Figure 1, for listening to the inter-CPU bus and detecting writes for invalidating the corresponding cache blocks in the own caches. Because the LEON3 processor does not support self-modifying code, snooping is required only for the data cache.

B. The Cache Mapping Design

Pivotal for our implementation of the reconfigurable cache mapping is the way reconfigurability is supported on Xilinx FPGAs. There are three approaches relevant for our application: partial reconfiguration, virtual reconfiguration and reconfiguration via shift registers.

Partial reconfiguration is the native reconfiguration approach supported by the Xilinx tools. A partial bitstream encodes the complete information of an FPGA region including the configuration of the switch boxes and LUTs. The bitstream is generated by the Xilinx tools and has a non-disclosed format. While others have shown that using reverse engineering parts of the bitstream can be decoded and, subsequently, be modified by custom tools, we have decided to avoid this reconfiguration type as the formatting of a partial bitstream is highly depended on the actual device and the place of the reconfigurable region on the FPGA.

Virtual reconfiguration denotes the implementation of the multitude of different functions in custom logic and clamping the function selection multiplexer to a memory bus. For CGP,

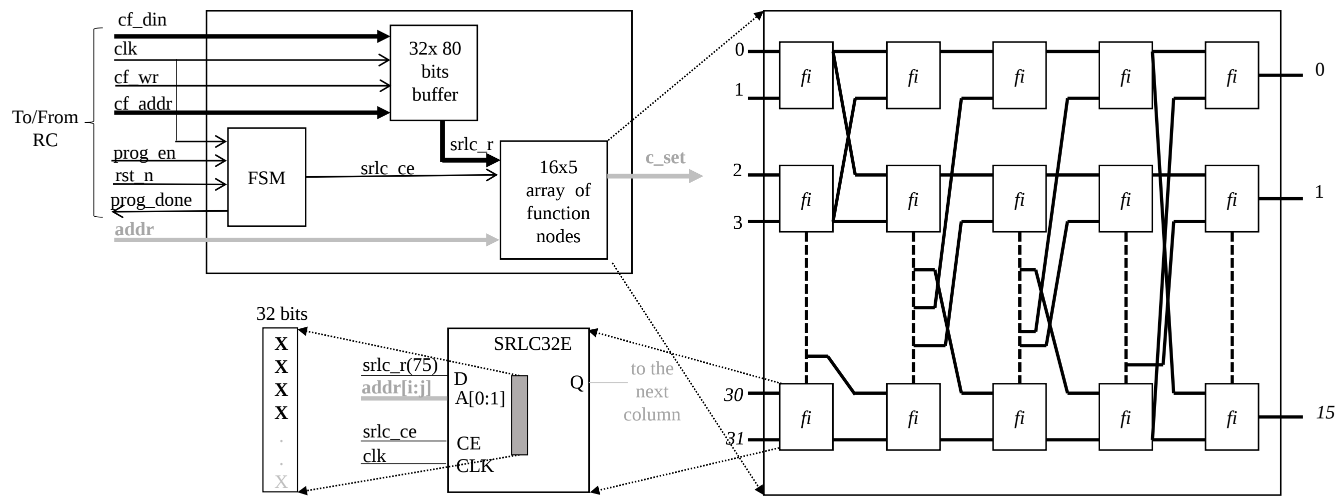


Fig. 2: The design of CGP-based cache mapping by using Xilinx's SRLC32E primitives as CGP blocks

this would mean implementing custom logic LUTs, each with a memory bus interface, in a two-dimensional grid. While virtual reconfiguration is the reconfiguration approach that leads to the smallest reconfiguration times, it is also very costly in terms of area overhead.

The third approach, and the one we have chosen, is reconfiguration via shift registers using the Xilinxs SRLC32E primitive that allows for specifying the content of some LUTs, which are then shifted in in a serial manner. While this approach is slower than virtual reconfiguration, it allows us to manipulate the native FPGA LUTs without using the Xilinx tools. The drawback is that switch boxes cannot be modified this way. Thus, routing has to be realized either by additional intermediate LUTs that are relaying signals according to the encoding of the solution or by fixing the routing and letting the evolutionary search process implicitly exploit it. Fixed routing works very well, especially when using the butterfly routing that easily allows for even distant address bits to be input to a single CGP functional block. The butterfly routing architecture is shown on the right-hand side of Figure 2. The size of our realized CGP architecture is 16 rows by 5 columns. Each of the functional blocks is a native Xilinx LUT. While the native Xilinx LUTs on a Virtex 6 FPGA have 5 inputs, only the first two inputs are used for CGP describing our address mapping function. We plan to use the full width of the native Xilinx LUTs in future work. The whole CGP mapping function has 32 primary inputs and 16 primary outputs. The inputs can be connected to any of the inputs of LUTs in the first column.

Figure 2, top of the left-hand side, details the CGP reconfiguration mechanism. The mechanism consists of an 80-word buffer with LUT contents and an FSM, shifting this buffer into the native Xilinx LUTs, which are chained. The reconfiguration controller (RC), which is not pictured in Figure 2, informs the FSM to start reconfiguration by setting the programming enable signal $prog\_en$. The FSM drives the chip select enable signal $srlc\_ce$ connected to all SRLC32E-configurable CGP LUTs and starts shifting in the buffer. At the end, the FSM acknowledges to the RC that the reconfiguration process was completed by pulsing the $prog\_done$ signal.

C. The Integration the into LEON3 Architecture

Figure 3 shows the overall integration into the LEON3 architecture. The architecture comprises a reconfiguration controller (RC), reconfigurable regions for each of the cache mappings and one cache hash function controller, denoted as Cache-HF Controller. All extensions to the standard LEON3 platform are colored in gray in Figure 3.

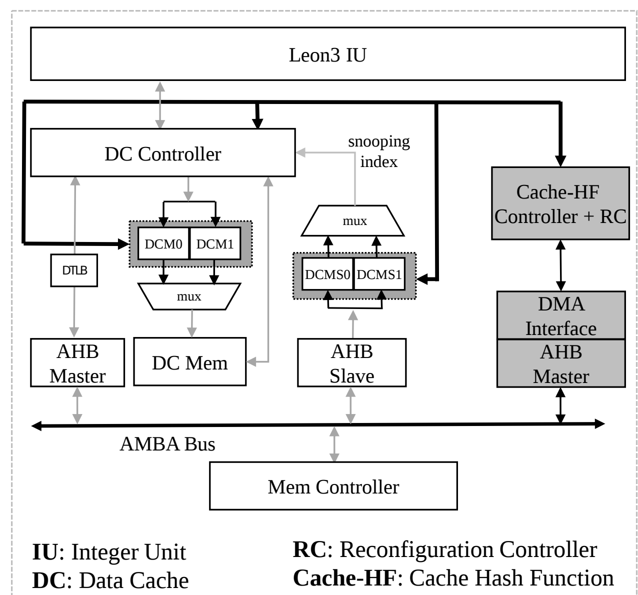
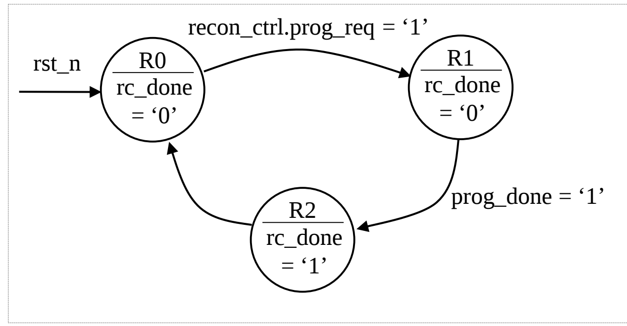


Fig. 3: The Integration into LEON3 SoC. IC mapping is not shown.

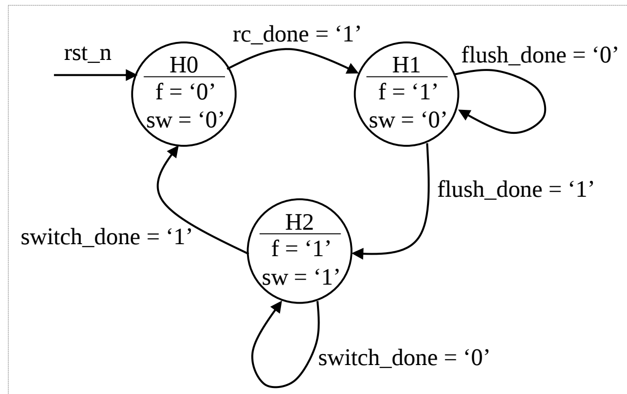
The RC works in cooperation with a DMA controller. This speeds up the transfer times of bitstreams located in the main memory to the reconfigurable regions of an FPGA. RC provides four main registers: the reconfiguration control register (`recon_ctrl`), that starts and stops the read and write transactions between the main memory and the reconfiguration area, the reconfiguration data length register (`recon_len`), indicating the size of the transferred data block, the reconfigurable data address register (`recon_addr`), specifying the physical memory address of the bitstream, and the reconfiguration status register (`recon_stat`), indicating the status of the reconfiguration process. In order to access registers of the RC, the Address Space Identifier (ASI) `lda/sta` instructions of the SPARC architecture are used [14]. These instructions are available in system mode only. Especially, the `ASI = 0x02` is reserved for system control registers and is used for interfacing the presented controllers. Since the `ASI = 0x02` is reserved for system control registers and has an unused address range from `0x10` to `0x1C`, this region is picked for interfacing with the RC.

Figure 4 (a) demonstrates the operation of the RC. In state $R0$, whenever there is the programming/reconfiguration request $prog\_req = 1$ raised in the `recon_ctrl` register by CPU, the reconfiguration process starts by a transition into $R1$. Once the reconfiguration is done, which is signaled by $prog\_done = 1$, the RC is signaling the end of reconfiguration by pulsing a "1" on the rc_done line by transiting to $R2$ for a single cycle and then back to $R0$. In state $R2$, the RC clears the $prog\_req$ bit in the `recon_ctrl` register and updates the `recon_stat` register.

The RC operates interleaved with the Cache-HF Controller, which in turn handles the two key cache mapping's operations: flushing cache memory once the reconfiguration process is done, and handling the cache mapping switches. Figure 4 (b) demonstrates the FSM of the Cache-HF Controller. Starting with the rc_done signal that is generated by RC to indicate that the reconfiguration process of the currently inactive mapping functions is finished, the Cache-HF Controller transits from $H0$ to $H1$ where it flushes the cache. Once the cache is empty, signaled by $flush\_done$, the Cache-HF controller moves to $H2$ switching currently inactive mapping functions active and vice versa. The switching mechanism finishes within one clock cycle, ensuring that the snooping protocol operations are not affected even in a multicore configuration.



(a) The simplified RC's FSM. $rc\_done = 1$ indicates the reconfiguration process done.



(b) The Cache-HF Controller's FSM. $f = 1$ and $sw = 1$ indicate $flushing$ and $switching$ processes happening.

Fig. 4: The FSM of RC and Cache-HF Controller's operations.

IV. Genetic Optimization

Based on the hardware implementation presented above, we are now explaining how to exploit the idea of natural selection in order to build better CPU caches. The strategy is to deploy an offline training phase by running an evolutionary algorithm finding reconfigurable cache mappings for an application and some application training input data vectors. In the subsequent testing phase, we validate the evolved mapping to see if the improvement generalizes for unknown data vectors and how the evolved mappings compare to the modulo cache mapping function.

Algorithm 1 shows our training algorithm for a set A of n applications. Each application a_i has $m = 4$ application training input data vectors, stored in I_i. For each application a_i the algorithm starts a loop where it initializes the first mapping function either by the conventional modulo mapping or randomly (line 2). The selection of these two initialization vectors is motivated by the observation in [9], where better results could be achieved by randomly initializing the initial solution. After evaluating the first solution (lines 3 and 4), the algorithm evolves 1000 iterations where in each iteration (lines 5 to 11) four offspring solutions are created (lines 6 to 9) by mutating (modifying four bits) the parent solution (line 7). The offspring solutions are evaluated (lines 8 and 9) and the best offspring solution becomes the new parent solution (lines 10 and 11) except for the case that all offspring solutions are worse than the parent solution. Finally, the output of the training algorithm is the set $F : \{f_0, ..., f_{n-1}\}$ containing the optimized cache mappings for all applications.

The functional quality of a candidate solution is evaluated (lines 3 and 4 as well as 8 and 9) by a LEON3 processor on an FPGA. For this, the cache mapping of a candidate solution is configured and the application is executed on four application input training vectors in I_i.

V. Experimental Results

We have synthesized the LEON3 system to a Xilinx ML605 Virtex-6 board. Table I shows the parameterization of the LEON3 system. Both level one caches are 2-way set associative and have a size of 8 kB. With 31 address bits and 32 bytes in an instruction cache block as well as 16 bytes in a data cache block, address bits [30:5] and [30:4] are inputs to instruction as well as data cache mapping functions, respectively. These address bits are also saved as tag bits to detect collisions. Seven and eight outputs bits from the cache mapping functions are used to index cache lines in the instruction as well as in the data cache, respectively. An example is shown in Figure 5.

We have selected four applications from the Mibench

Algorithm 1: `The Training Algorithm`

Input: $A : \{a_0, ..., a_{n-1}\}$ - n applications
Input: $I : \{I_0, ..., I_{n-1}\}$ - n input data sets, $|I_i| = m$
Input: f_{init}: f_{mod} or f_{rand} - the first mapping
Output: $F : \{f_0, ..., f_{n-1}\}$ - n optimized cache mappings

for *each* a_i *in* A **do**
  $f_{parent} \leftarrow f_{init}$
  $chf\_recon(f_{init})$
  $T_{parent} \leftarrow exec(a_i, I_i)$
  for *each generations in* 1000 **do**
    for *each child_j in* 4 **do**
      $f_{child\_j} \leftarrow mutate(f_{parent})$
      $chf\_recon(f_{child\_j})$
      $T_{child\_j} \leftarrow exec(a_i, I_i)$
    $T_{parent} \leftarrow min(T_{parent}, T_{child\_0}, ..., T_{child\_3})$
    $f_{parent} \leftarrow update\ the\ best$
  $f_i \leftarrow f_{parent}$

TABLE I: LEON3 platform and system configuration

System Configurations	Description
Clock Frequency	75Mhz
Integer Unit	Yes
Floating Point	Software
Instruction Cache	2-way associative, 8KB, 32bytes/line, LRU
Data Cache	2-way associative, 8KB, 16bytes/line, LRU
Memory	1GB DRAM

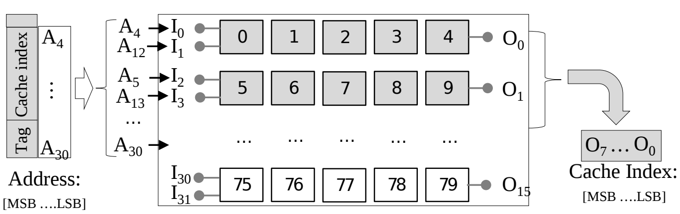


Fig. 5: An example mapping of address bits to CGP inputs for a data cache. The mapping connects $A4 \rightarrow I0, A5 \rightarrow I2$, and so on, which allows for initializing a modulo mapping.

benchmark: SHA, QSORT, DIJKSTRA and FFT. Four input data vectors have been created randomly. To simplify the experiment evaluation, we have executed the experiments without an operating system to avoid context switches. For each application, we have started the optimization run from a randomly initialized solution and from the modulo cache mapping function. Also, we have tried all possible optimization configurations, i.e., optimizing only the instruction cache, only the data cache and both cache mapping functions at the same time. Table II summarizes those configurations.

TABLE II: Different configurations in the training phase

Training configurations	Description
$L1 : I - Mod., D - Opt.$	IC mapping is modulo, DC mapping is optimized
$L1 : D - Mod., I - Opt.$	DC mapping is modulo IC mapping is optimized
$L1 : I, D - Opt.$	Both IC, DC are optimized
Mibench Applications: $n = 4$	SHA, QSORT, DIJKSTRA, FFT
The number of input data $m = 4$	For each application, the input data is generated randomly

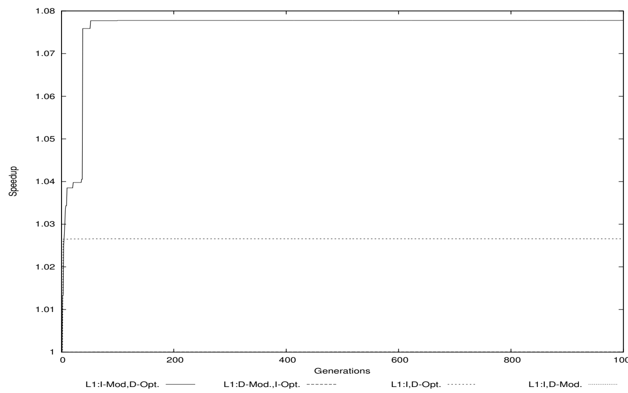
The developments of the execution times relative to a conventional cache are shown in Figure 6 for all benchmarks and both initialization methods. In the first line (Figure 6 (a)) the results for the SHA benchmark are presented. There, when starting from a modulo cache mapping and optimizing only the data cache mapping, the evolution achieves 7.8% improvement in execution time. An improvement of 2.7% is possible if starting from a modulo cache mapping function and optimizing both cache mappings. If the training algorithm initializes the first mapping randomly (Figure 6 (b)), only when optimizing both cache mapping an improvement of about 2.7% can be achieved. In Figure 6 (c), the FFT training results are shown. Improvements of about 1.9% for L1:I and 1.8% for L1:I,D are possible, if starting from modulo mappings. However, no improvements can be observed when starting from a modulo cache mapping function (Figure 6 (d)).

While the execution time have been improved for SHA and FFT during the training experiments, no such cache mappings have been found for QSORT and DIJKSTRA (Figure 6 (e), (f), (g), (h)).

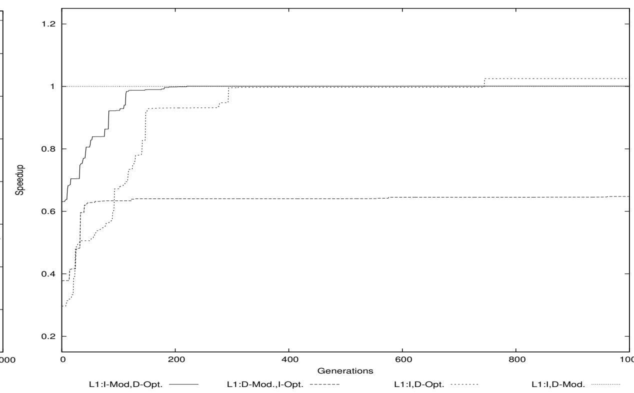
Next, we test the applications by executing them using the evolved cache mappings on input data vectors that have not being used during the training. As shown in Table III, there are 7.9% and 2.4% improvements for SHA, optimizing $L1 : D$ and both $L1 : I, D$ mappings if the first mapping is started from modulo, and we see only 2.5% improvement in case of optimizing both $L1 : I, D$ mapping if the mapping is initialized randomly. There are 1.9% and 1.8% improvements for optimizing FFT if the first mapping is initialized with modulo only.

VI. Conclusions and Future Work

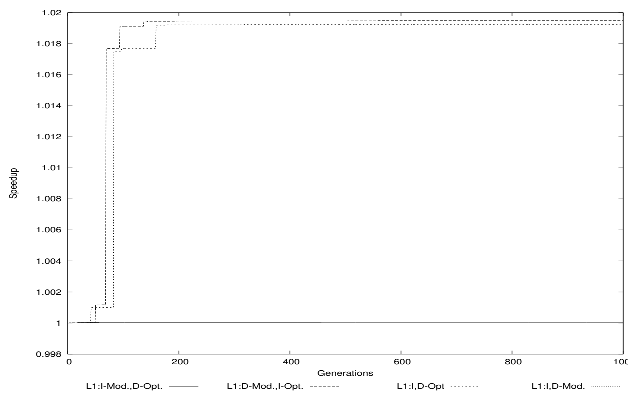
In this paper we have presented for the first time an implementation of a system with dynamically reconfigurable cache mapping functions and used this system to evolve optimized cache mappings as well as test their generalization behavior. We have found that for the SHA and FFT benchmarks there are better cache mappings rather than the traditional modulo-based cache mapping function, able to speed up the execution time by 7.9% and 1.9%, respectively. We will extend our work to a multi-core scenario and cover more benchmarks. Furthermore, our reconfiguration and measurement framework is suitable for further investigations on optimization of caches



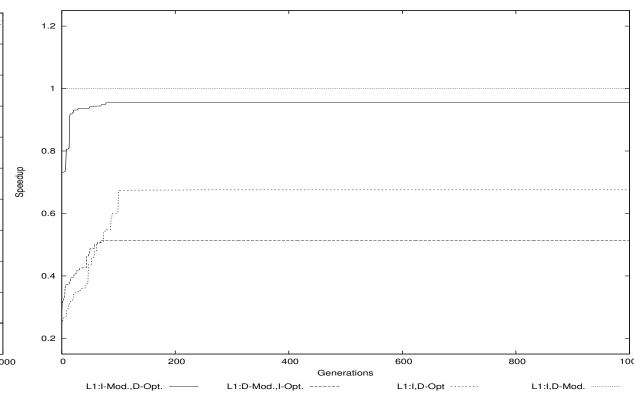
(a) SHA - CGP is initialized with modulo



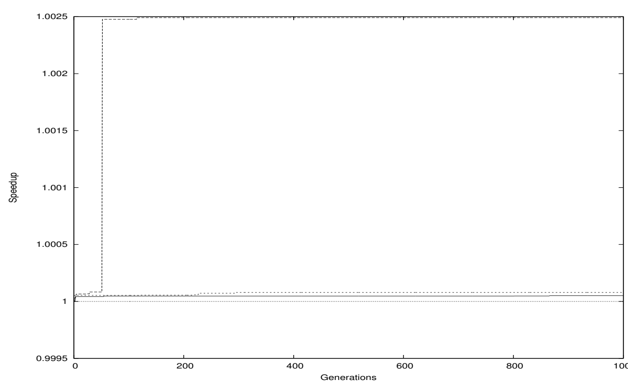
(b) SHA - CGP is initialized randomly



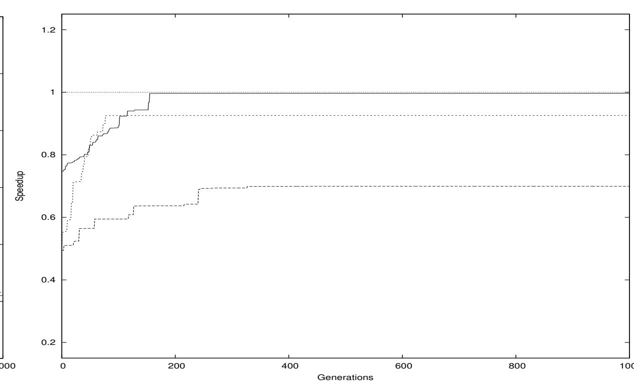
(c) FFT - CGP is initialized with modulo



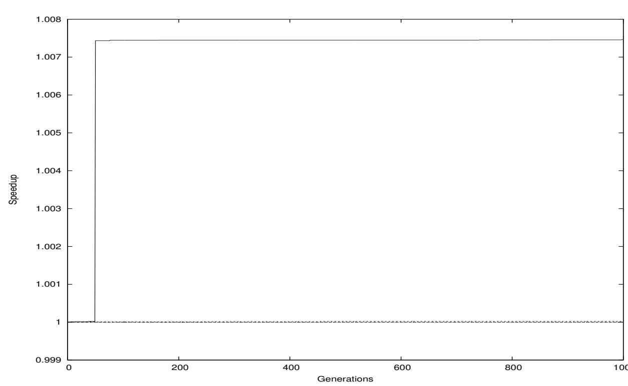
(d) FFT - CGP is initialized randomly



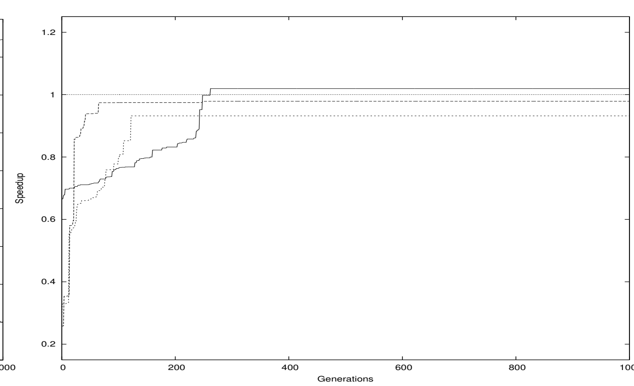
(e) QSORT - CGP is initialized with modulo



(f) QSORT - CGP is initialized randomly



(g) DIJKSTRA - CGP is initialized with modulo



(h) DIJKSTRA - CGP is initialized randomly

Fig. 6: The result of training phase; $L1 : I, D - Mod.$: IC and DC mapping are modulos; The speedup is calculated by normalized to execution time of $L1 : I, D - Mod.$ case.

TABLE III: Performance improvement (in %) achieved in the testing phase.

CGP initialized with a modulo mapping			
	L1:D-Mod, L1:I-Opt.	L1:I-Mod., L1:D-Opt.	L1:I, D-Opt.
QSORT	0.06	0.02	0.014
SHA	0.00	7.90	2.40
FFT	1.90	0.01	1.88
DIJKSTRA	0.00	0.64	0.00
CGP initized randomly			
	L1:D-Mod, L1:I-Opt.	L1:I-Mod., L1:D-Opt.	L1:I, D-Opt.
QSORT	-19.91	-0.27	-5.66
SHA	-35.37	0.00	2.53
FFT	-48.67	-12.11	-32.39
DIJKSTRA	-1.46	-10.12	-6.39

and memory interfaces, which we are planning to do in the future.

References

[1] D. A. P. John L. Hennessy, *Computer Architecture A Quantitative Approach*. Elsevier, 2012.

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

[3] 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.

[4] A. Gordon-Ross and F. Vahid, "A Self-Tuning Configurable Cache," in *Design and Automation (DAC)*. IEEE, 2007, pp. 234–237.

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

[7] 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.

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

[9] P. Kaufmann, C. Plessl, and M. Platzner, "EvoCaches: Application-specific Adaptation of Cache Mappings." IEEE CS, 2009, pp. 11–18.

[10] P. Kaufmann and M. Platzner, "Multi-objective Intrinsic Evolution of Embedded Systems," in *Organic Computing — A Paradigm Shift for Complex Systems*, ser. Autonomic Systems, C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Springer Basel, 2011, vol. 1, pp. 193–206. [Online]. Available: http://dx.doi.org/10.1007/978-3-0348-0130-0_12

[11] L. Sekanina, J. A. Walker, P. Kaufmann, C. Plessl, and M. Platzner, "Evolution of Electronic Circuits," in *Cartesian Genetic Programming*, ser. Natural Computing Series. Springer Berlin Heidelberg, 2011, pp. 125–179. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17310-3_5

[12] P. Kaufmann, *Adapting Hardware Systems by Means of Multi-Objective Evolution*. Berlin: Logos Verlag, 2013. [Online]. Available: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=3530&lng=deu&id=

[13] ——, "Multikriterielle Evolution adaptiver eingebetteter Systeme," *Ausgezeichnete Informatikdissertationen, GI-Edition - Lecture Notes in Informatics (LNI), Series of German Informatics Society, Springer*, vol. D-14, pp. 71–80, 2014.

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

[15] J. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP)*. Springer LNCS, 2000, pp. 121–132.