Towards Self-Adaptive Caches: a Run-Time Reconfigurable Multi-Core Infrastructure

Nam Ho
University of Paderborn, Germany
Email: namh@mail.upb.de

Paul Kaufmann
University of Paderborn, Germany
Email: paul.kaufmann@gmail.com

Marco Platzner
University of Paderborn, Germany
Email: platzner@upb.de

***Abstract*—This paper presents the first steps towards the implementation of an evolvable and self-adaptable processor cache. The implemented system consists of a run-time reconfigurable memory-to-cache address mapping engine embedded into the split level one cache of a Leon3 SPARC processor as well as of an measurement infrastructure able to profile microarchitectural and custom logic events based on the standard Linux performance measurement interface `perf_event`. The implementation shows, how reconfiguration of the very basic processor properties, and fine granular profiling of custom logic and integer unit events can be realized and meaningfully used to create an adaptable multi-core embedded system.**

I. Introduction

One of the major application areas of Field Programmable Gate Array (FPGA) devices is the pre-production development and testing of new integrated circuit designs. System architects can significantly reduce time-to-market when compared to the more time and cost demanding procedure using the creation of intermediate test Application Specific Integrated Circuits (ASIC). With the rising chip sizes, however, prototyping with FPGAs suffers from the notoriously time demanding electronic design automation (EDA) tool chains making the developers statically partition their designs and resynthesizing only the modified chip areas. When using Xilinx' tools [1], partial synthesis is supported to a large extent, allowing to avoid full system resynthesis and to reconfigure FPGA segments on-the-fly by partial bitstreams through the Internal Configuration Access Port (ICAP).

Besides shorter synthesis times, dynamic reconfiguration can also be used for adaptable systems. Functions that have to change their behavior over time can be presynthesized in different flavors, according to the estimated adaptation scenarios, or can be generated on the fly. Examples for microarchitectural adaptation are the self-tuning reconfigurable cache system presented in [2] and the reconfigurable cache mappings described in [3]. Additionally, using FPGAs for accelerating computing tasks is emphasizing the integration of reconfigurable hardware parts as coprocessors into a conventional architecture. This has shown providing high performance computing capabilities even for general-purpose embedded processors [4], [5].

However, the combination of reconfigurable fabric parts inside conventional processor architectures is challenging. The need for efficiently reconfigurable computing platforms has created multiple contributions spanning specialized programming [6], [7] and architecture model areas [8], [9], [10], [11]. For instance, the ERA project [11] investigates synthesis tools and hardware design aspects for the realization of an efficiently reconfigurable platform for embedded systems. Using a dedicated reconfiguration controller, the ERA system is able to dynamically adapt instruction-sets, register files, Network on Chip (NoC) interconnects and memory hierarchies. Xilinx' recent System on Chip (SoC) Zynq-7000 FPGA [12] provides flexible ways for extending conventional processors by reconfigurable functions. In the context of Evolvable Hardware research [13], [14], [15], the work in [16] shows the implementation of evolvable circuits for image filtering, exploiting virtually reconfigurable circuits and dynamic partial reconfiguration.

Unfortunately, the work on adaptable processors faces many technical difficulties such as the inefficient bitstream management and lack of performance monitoring features [17]. Both are required for self-adaptive processor designs. The architecture presented in this paper supports both, a performance monitoring infrastructure, which is especially designed for profiling of the underlying microarchitecture aspects at run-time, and an abstraction layer for managing reconfiguration data by providing built-in Reconfiguration Controller (RC) accompanied by a Linux device driver.

In the remainder of the paper, we first describe the concept of an evolvable cache (Section II), then the performance measurement architecture in Section III. In Section IV, we present experimental results for the performance measurement infrastructure using MiBench workloads, and show processor reconfiguration on the example of on-the-fly cache mapping function variation. Section V concludes the paper, outlining future work.

II. The EvoCache Concept

Inside microprocessors, caching techniques play an important role to hide the latency of the main memory access. Caching introduces a hierarchy of intermediate memories that level the access latencies between the slower main memory and the faster registers of a CPU. As closer the cache memories are in the hierarchy to the CPU registers, as faster and smaller they get. The main memory addresses are mapped to the address spaces of intermediate cache memories by computing the modulo function on the main memory address. As modulo factor, the corresponding cache size is selected. Conventional systems are using the described memory-to-cache mapping function because it has no temporal and resource overhead. However, one can imagine having multiple memory-to-cache address mapping functions tailored to different applications resulting in better execution times. One of the promising methods to find better cache mappings is to exploit the techniques of Evolvable

Hardware for the optimization of hardware by evolutionary algorithms. In [3], [18], [19], [20], [21] first work on such a system, coined EvoCache, was presented. In this case study, we show how our proposed architecture enables to deploy and evaluate the EvoCache idea directly on a reconfigurable hardware platform.

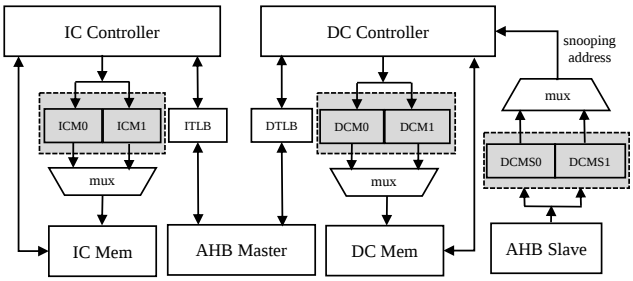

Fig. 1: Reconfigurable cache mapping. Abbreviations: IC/DC = instruction/data cache; {I/D}M{0/1/S0/S1} = instruction / data / cache mapping function {0/1/snooping 0/snooping 1}

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. As a multi-core system with distributed caches is targeted, for each CPU redundant reconfigurable fabrics snoop the inter-CPU bus and help detecting write back and write trough collisions. Fig. 1 presents the architecture in which, the gray parts are partial reconfigurable fabrics dedicated as the mappings for instruction/data cache. For virtually addressed caches the architecture needs to be extended by an additional collision unit, not presented in Fig. 1.

An evolutionary algorithm (EA) does the optimization of cache mappings in two phases. As indicated in Fig. 2d, the EA takes an application and optimizes an according memory-to-cache address mapping function regarding the execution time iteratively and off-line. The optimization can also take other metrics, such as the miss rates and energy consumption, and be done on the fly. This, however, is future work.

Candidate solutions for memory-to-cache address mapping functions are encoded using the Cartesian Genetic Programming model (CGP) [22]. CGP is well suited to represent combinational logic circuits as it encodes a two dimensional grid of functional nodes connected by feed forward wires. Mapping of CGP encoded circuits to an FPGA can be done in many ways. In this work we, map CGP nodes to neighboring 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. This is presented in Fig. 2a. The final architecture is therefore quickly reconfigurable, as only FPGA LUT contents need to be changed, and has a compact footprint. The LUTs in our case study are fixed to have 2 inputs and can implement up to 16 different functions each (Fig. 2b).

Fig. 2a shows an example of a reconfigurable cache function with 8 inputs and 4 outputs. By fixing the inner routing, the searching space gets constrained. In order to allow the evolutionary algorithm for a more general search space, address bits may be permuted freely before connected to the inputs of the functional nodes in the first column of the CGP model. Fig. 2d depicts an encoded chromosome. As the CGP routing is fixed, the chromosome encodes only the LUT functions, starting with the LUT in the upper left corner, followed the next LUT in the column, and continuing column-wise. The outputs of the last LUTs define the global outputs. As the Leon3 platform supports cache sizes of up to 256kB/way, 16 bits are required to encode a cache line with minimum 4 bytes / line in a direct-mapped cache. Table I shows the hardware resource usages for the CGP implementation, having 32 inputs, 16 outputs, synthesized and generated with the partial reconfiguration tool flow from Xilinx for Virtex-6.

TABLE I: CGP implementation

	Number LUTs used reported by `ise`	Partial bitstream size `.bit` reported by `bitgen`
CGP nodes implemented as 5-LUTs primitives	80	228647 bytes

In order to make EvoCache work with our proposed architecture, we have allocated two partially reconfigurable functions for each 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 switch to the new mapping. As showed also in Fig. 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), excepted it needs more two mappings for supporting snooping operations, DCMS0/1 (Data Cache Mapping for Snooping). The reason is that in snooping protocol, for the same address the cache line indexes used to check invalidation would be consistent with the one issued by CPU. That means the cache mapping to data cache memory has to be the same in both cases of deriving addresses from snooping operation and from CPU issued.

III. The Proposed Architecture

Fig. 3 shows the overall baseline architecture. The architecture composes of up to four Leon3 soft-core processors [23], a reconfiguration controller (RC) connected to one of the Leon3 processors, reconfigurable regions, and performance monitoring units connected to each of the Leon3 processors. All extensions to the standard Leon3 platform are colored in gray.

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. The reconfiguration regions can realize custom logic that may also connect to the main AMBA bus. The Performance measurement Units (PMU), one for each Leon3 core, integrate with the main interrupt controller and are able but not limited to monitor CPU cycles, cache misses, TLB misses, and reconfiguration times. In order to access registers of PMUs and the RC, the Address Space Identifier (ASI) `lda/sta` instructions of the SPARC architecture are used [23]. 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. The following sections describe the implementations more detailed.

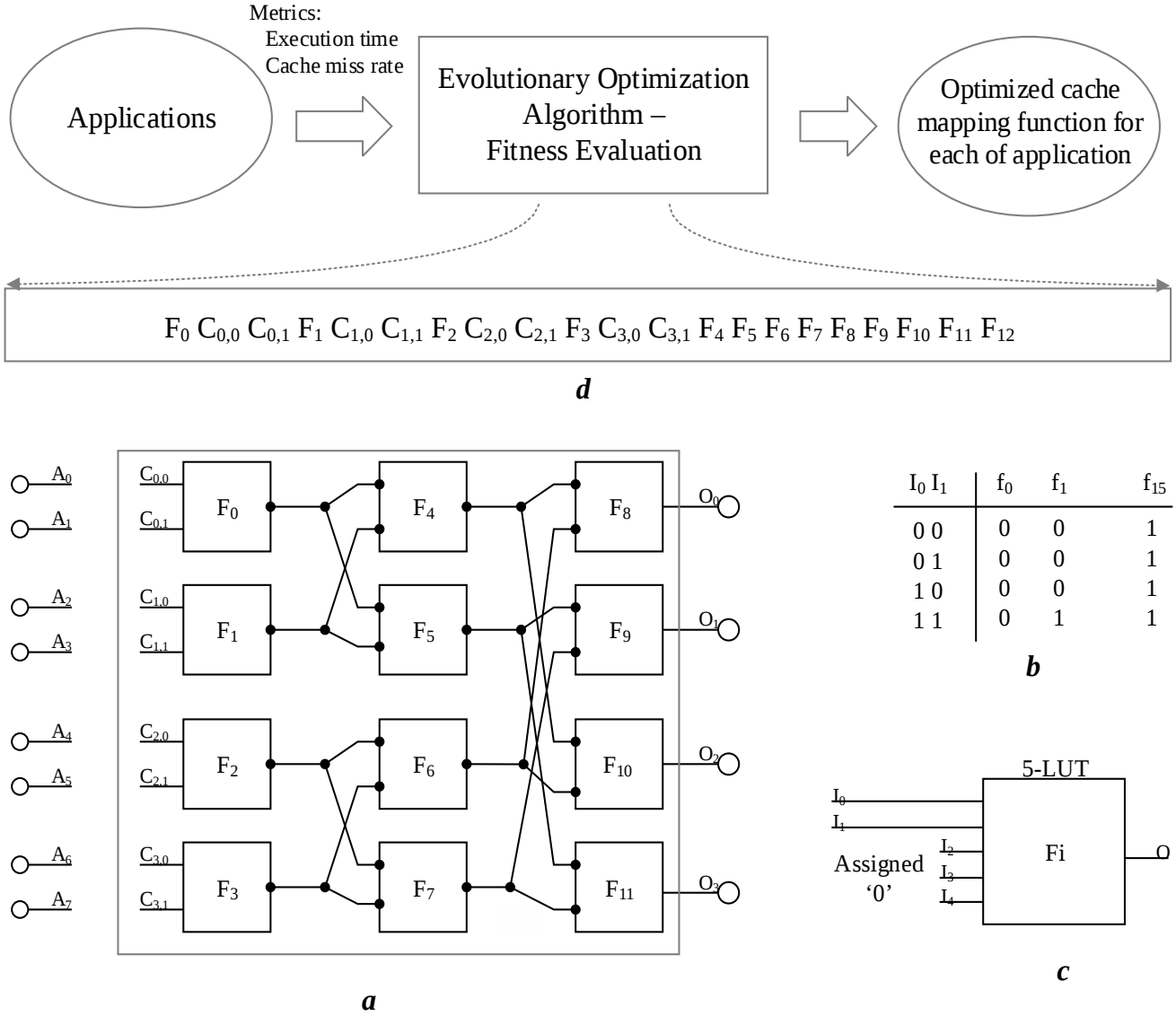

Fig. 2: An example of adaptive CGP module for a cache mapping with 8 inputs, 4 outputs by fixing the routing. (a) The circuit designed in details such that each of CGP nodes is a 2-LUT. (b) The 2-LUTs can be reconfigured with 16 functions. (c) On Virtex-6, 2-LUTs implemented as CGP nodes by using 5-LUTs. (d) The encoded chromosome is reduced size by removing inter-connections encoded parts.

A. Reconfiguration Controller

The heart of the proposed architecture is the RC, showed in Fig. 4. It 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. The memory map of the registers in presented in Table II. 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 the RC.

Fig. 5 shows the main finite state machine (FSM) of the RC, `Intf_Ctrl_Block`. Inside the RC, `Intf_Ctrl_Block` is responsible for interacting the master CPU. Setting `recon_ctrl(0)` to '0' and `recon_ctrl(31)` to '1' starts reconfiguration. The RC fetches bitstream data from memory pointed by the address register and writes it via the asynchronous FIFOs to the `Recon_Ctrl_Block` module. Three asynchronous FIFOs are reserved for communication between the `Intf_Ctrl_Block` and the `Recon_Ctrl_Block`, those are: `fifo_ctrl` is transmitting commands to the `Recon_Ctrl_Block`; `fifo_data` is sending bitstream data from the `Intf_Ctrl_Block` to the `Recon_Ctrl_Block`; and `fifo_fb` transmits feedback commands from the `Recon_Ctrl_Block` to the `Intf_Ctrl_Block`.

`Recon_Ctrl_Block` in turn reads the commands from

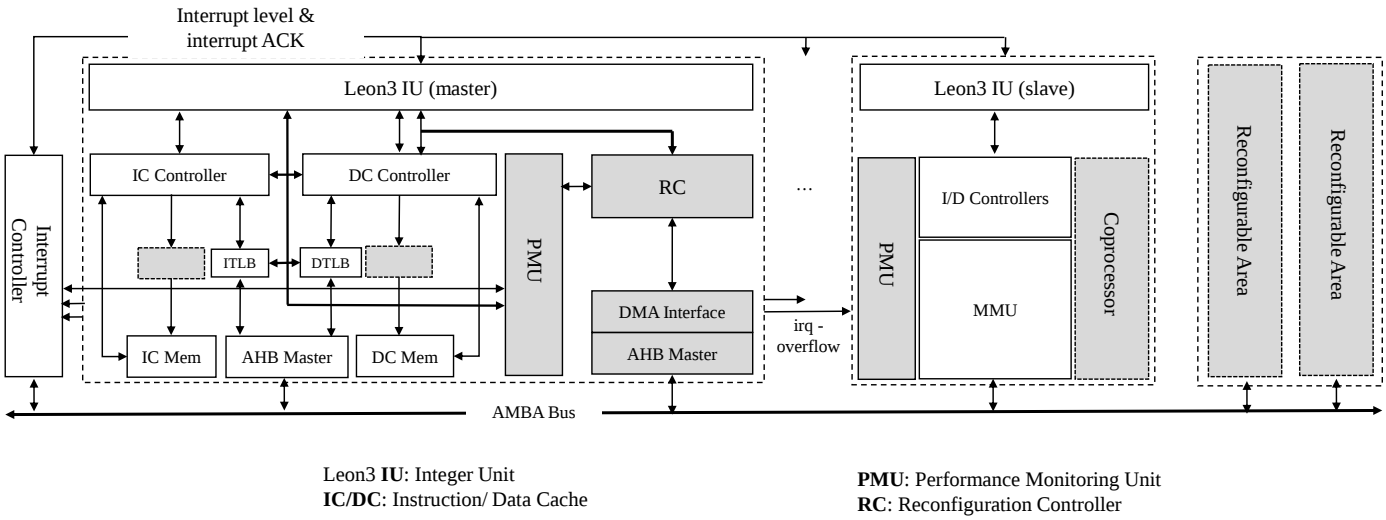

Fig. 3: The proposed baseline of reconfigurable architecture.

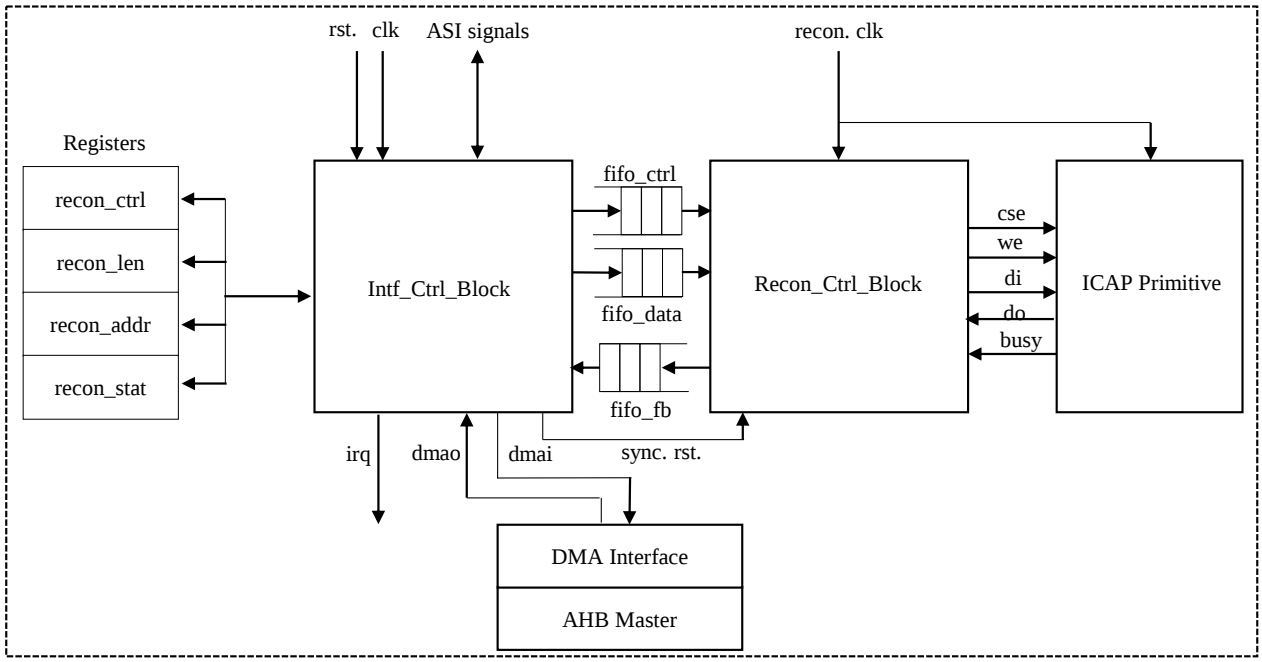

Fig. 4: The reconfigurable controller with a DMA interface.

`fifo_ctrl` and either reads back partial bitstream or program one of the reconfigurable fabrics. Its main FSM is depicted in Fig. 6. For example, if the `Recon_Ctrl_Block` gets the `RECON_PROG` command indicating reprogramming in state `RECON_CTRL_READ_CMD`, it jumps to a reconfiguration state, `RECON_CTRL_PROG`, reads the reconfiguration data from the `fifo_data` queue, and reconfigure the FPGA via the ICAP interface. When receiving `RECON_PROG_END` command, `Recon_Ctrl_Block` finishes the reconfiguration process and sends back `RECON_FB_DONE` command via the `fifo_fb` queue.

During the reconfiguration process, `Recon_Ctrl_Block` can do a quick read back of the programmed bitstream for error checking. Should a disparity of the programmed and read back bitstream occur, `RECON_FB_ERR` is send to `Intf_Ctrl_Block`, which in turn updates the `recon_stat` register and triggers an interrupt of the master CPU. The handling of configuration errors is done in the `RECON_FEEDBACK_READ_DONE` state of `Intf_Ctrl_Block`.

At the software side, a device driver enumerated at `/dev/reconctrl` is implemented for interfacing the RC. By that abstraction, the user has just to allocate a buffer for bitstream data, which is then handled by that driver automatically.

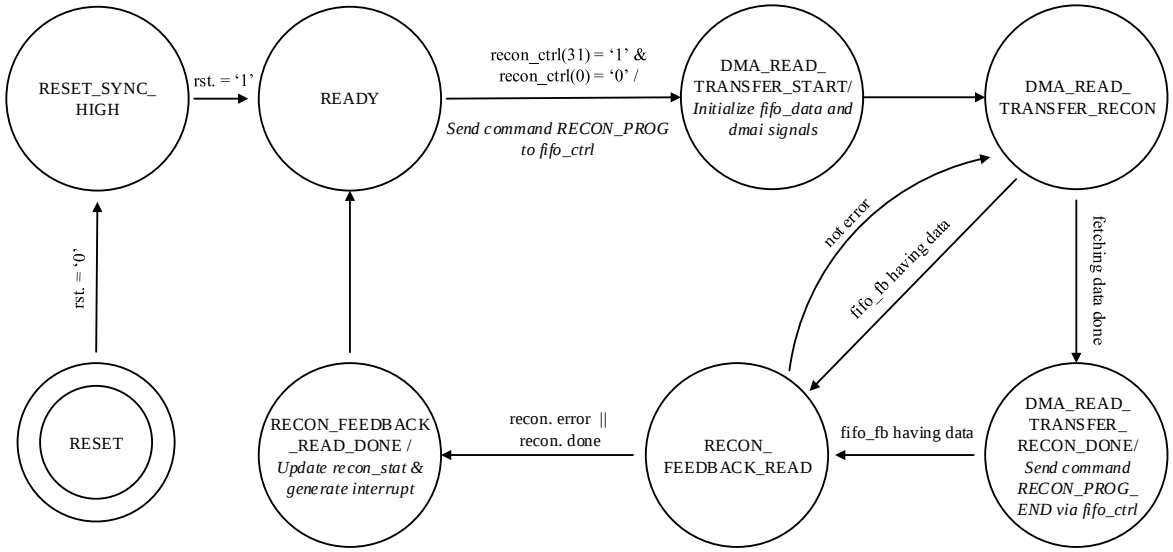

Fig. 5: The main Finite State Machine (FSM) of Intf_Ctrl_Block.

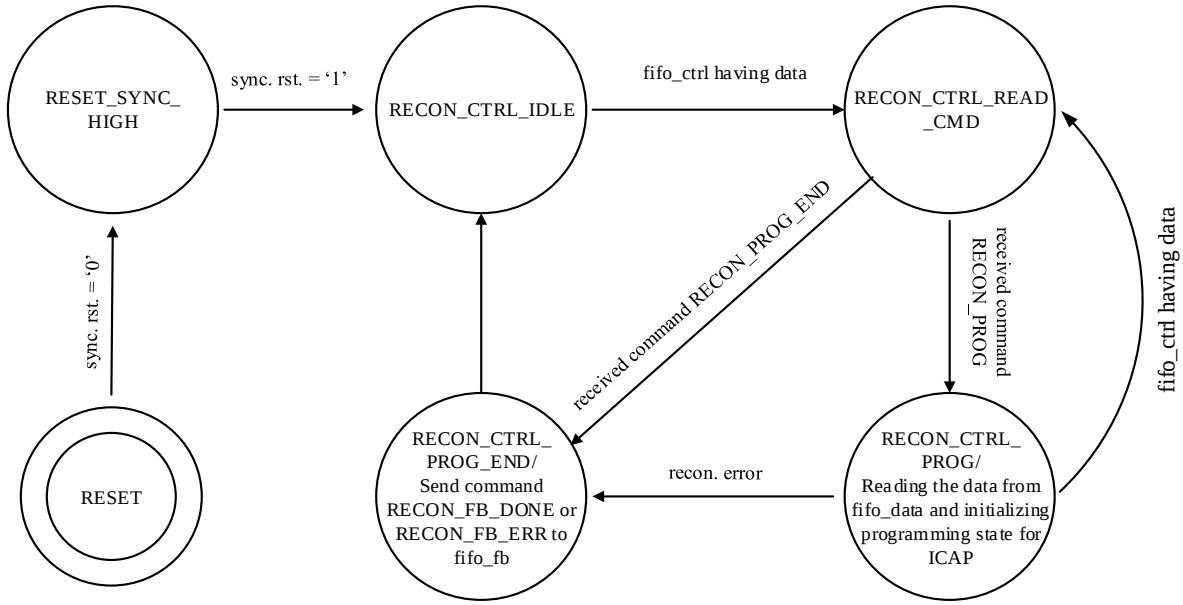

Fig. 6: The main Finite State Machine (FSM) of Recon_Ctrl_Block.

B. Infrastructure for Performance Monitoring

We have been tailoring PMUs for the Leon3 multi-core platform. To be able to handle the performance counters properly for workloads that are migrating between the processors, the PMUs have been replicated for each processor core and the performance counters are stored to and are loaded from the context of an application by the OS kernel during each context switch. Fig. 3 shows the PMU placement in the current implementation. The PMUs are connected to the signals driven out from the Integer Units (IU), from L1 instruction and data cache controllers, and from the I-TLB as well as the D-TLB modules of the MMU. Besides conventional processor events, reconfiguration times can also be measured by profiling the RC.

Table III shows the address mapping for the overall PMU system. The PMUs are connected to the memory bus in the same way as the RC is. The ASI address range of PMUs is `0xC0-0xFC`.

From the operating system side, we modified `perf_event`, the current standard performance monitoring software inside the Linux kernel, to handle PMUs. From the user space perspective, the `perf tool` is used for measurements. The `perf_event` interface and the `perf tool` work together as follows: The `perf tool` invokes an application for measurement. Depending on the input parameters, the `perf tool` provides the event sources to

TABLE II: Memory map of the RC registers.

Registers accessed via ASI = 0x02	
Register—32 bits	ASI Address Mapping
Reconfiguration control - recon_ctrl (RW): - Bit[0]: read/write - Bit[30..1]: reserved - Bit[31]: activate / deactivate reconfiguration process	$0x10$
Bitsream address - recon_addr (RW): - Bit[31..0]: physical address of bitstream data for DMA transfer	$0x14$
Bitstream length - recon_len (RW): - Bit[31..0]: length of bitstream data for DMA transfer (in words)	$0x18$
Reconfiguration status - recon_stat (R): - Bit[0]: error/success - Bit[30..1]: reserved - Bit[31]: reconfiguration done	$0x1C$

TABLE III: Memory map of the PMU system. The maximum number of event counters is limited to 7 due to the current limitations of our system design. The maximum number of monitored events can be up to 256. Currently, cycles, instructions, L1:I / L1:D access and read misses as well as L1:D write accesses and misses, and ITLB as well as DTLB misses are supported.

Registers accessed via ASI = 0x02	
Register—32 bits	ASI Address Mapping
Global control (RW): - Bit[0]: enable all event counters (en) - Bit[1]: reset/clear all event counters (rst) - Bit[2]: reset/clear cycles countered (cyc.rst) - Bit[7..3]: number of event counters supported - Bit[31]: reset/clear IRQ pending	$0xC0$
overflow status (RW): - Bit[0]: overflow cycle counter - Bit[n..1]: overflow for event counter - n..1 - Bit[31]: indication for IRQ pending	$0xC4$
Cycle counter (RW): - Bit[31..0]: counter value is being monitored	$0xC8$
Cycle counter control (RW): - Bit[7..0]: reserved - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	$0xCC$
The i^{th} event counter (RW): - Bit[31]: counter value is being monitored	$0xD0 + 8 \cdot (i^{th})$
The i^{th} event counter control (RW): - Bit[7..0]: event identifier (event_id) - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	$0xD4 + 8 \cdot (i^{th})$

monitor to kernel's `perf_event` measurement infrastructure. The `perf_event` infrastructure, in turn, configures the PMU infrastructure and starts profiling. When the application finishes its execution, the `perf tool` reads out the event counters and aggregates the final results via the `perf_event` interface. An example for the output of the `perf tool` is given in Fig. 7, printing out from our experimental platform, Leon3.

IV. Preliminarily experimental results

This section reports on our preliminarily experimental results for the reconfigurable application of evolvable cache

```
# perf stat -e cycles,instructions,
L1-dcache-loads,L1-dcache-load-misses,
L1-dcache-stores,L1-dcache-store-misses
./queens -c 14

  14 queens on a 14x14 board...
  ...there are 365596 solutions

Performance counter stats for './queens -c 14':
8295258591     cycles
5309555048     instructions     # 0.640 IPC
965961200      L1-dcache-loads
50932          L1-dcache-load-misses
191890081      L1-dcache-stores
27365021       L1-dcache-store-misses
115.170000000  seconds time elapsed
# _
```

Fig. 7: Example: Launch and output of `perf tool`.

mappings. The system configuration for the LEON3 platform is shown in Table IV. We have synthesized the LEON3 platform and programmed it to a Xilinx ML605 Virtex-6 board. The root file system is located on a CF card.

TABLE IV: Leon3 platform and system configuration

Clock Frequency	75Mhz
Integer Unit	Yes
Floating Point	Software
Instruction Cache	2-way associative, 8KB, 32bytes/line, LRU
ITLB	8 entries
Data Cache	2-way associative, 8KB, 16bytes/line, LRU
DTLB	8 entries
MMU	Yes
MP IRQ Controller	External IRQ (EIRQ) = 14
PMU	New feature supported
Linux Kernel	2.6.36.4 patch from Gaisler
Compiler	pre-built Linux toolchain from Gaisler

To demonstrate the complete architecture, we have set up experiments using four different Mibench test functions and measured their performances for different cache mappings. The simulations have been done using a single-core system only and all experiments have been repeated ten times. The results are presented in Table V.

In the first experiment the reference behavior of a regular system with a conventional cache (or mapping by the modulo function) has been measured. In the second experiment the least significant address wires have been swapped while indexing the caches. That is, for the instruction cache the sixth and seventh address bits have been mapped to the second and first cache index bits. For the data cache, the fifth and sixth address bits are mapped for the second and first cache index bits. In the third experiment, we have simulated the unusual case of having only two cache lines. All memory addresses are mapped according to the sixth and fifth address bits to two cache lines of the instruction and data caches, respectively.

The first observation is that the instruction counts have been measured accurately for all experiments. While this seems to be a requirement rather than an achievement, a measurement infrastructure integrated into a system using caches tends to suffer from small jitter that depends on the context switching

TABLE V: Statistical data collected for a subset workloads of MiBench by dynamically reconfiguring different cache mappings. Results compared by running `perf tool` 10 times on the same application, on one core platform.

	Exp. 1: Conventional cache				**Exp. 2: Swapping LSB cache index bits**			**Exp. 3: Using only 2 cache lines**		
Benchmark	Inst.	IPC	L1:I misses [%]	L1:D misses [%]	IPC	L1:I misses [%]	L1:D misses [%]	IPC	L1:I misses [%]	L1:D misses [%]
basicmath	9.93E+08	0.596	2.47	0.91	0.596	2.47	1.01	0.184	38.38	52.48
jpeg	3.61E+07	0.567	0.62	17.26	0.567	0.62	17.30	0.191	27.64	79.77
Dijkstra	5.75E+07	0.486	0.79	13.46	0.487	0.79	13.46	0.114	49.26	60.07
FFT	1.18E+09	0.599	2.06	0.72	0.598	2.06	0.73	0.153	47.46	59.86

mechanism of the operation system and the prefetched cache lines.

The next observation is that the cache misses for the conventional cache and the cache mapping with swapped least significant bits (Experiment 2) are almost identical for the instruction and only slightly different for the data cache. An explanation for this behavior can rely on the fact that the instruction cache fetches the same sequence of memory entries for all experiment repetitions while the data cache access pattern may also depend on the initial state of the pseudo random number generator used in the applications.

And the last observation is that if using only two cache lines (Experiment 3), the miss rates for both caches increase dramatically, as expected.

V. Conclusions and Future Work

In this paper, we have described a reconfigurable architecture and a performance measurement infrastructure for building adaptable processors. We have demonstrated that our system can be executed using different cache mapping functions and that the system can be measured precisely. Based on these results, next steps of our work are the incorporation of an optimization algorithm into the system and the online adaptation of cache mappings.

VI. Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no 257906.

References

[1] Xilinx, "Partial Reconfiguration User Guide." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

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

[3] P. Kaufmann, C. Plessl, and M. Platzner, "EvoCaches: Application-specific Adaptation of Cache Mappings," in *Adaptive Hardware and Systems (AHS)*. IEEE CS, 2009, pp. 11–18.

[4] M. A. Watkins and D. H. Albonesi, "ReMAP: A Reconfigurable Architecture for Chip Multiprocessors," *IEEE Micro*, vol. 31, no. 1, pp. 65–77, 2011.

[5] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *International Symposium on Computer Architecture (ISCA)*, 2000, pp. 225–235.

[6] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, pp. 1–33, 2009.

[7] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model," *IEEE Trans. VLSI Syst.*, vol. 21, no. 12, pp. 2179–2192, 2013.

[8] V. Lai and O. Diessel, "ICAP-I: A Reusable Interface for the Internal Reconfiguration of Xilinx FPGAs," in *Field-Programmable Technology (FPT)*. IEEE, 2009, pp. 357–360.

[9] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 234–239.

[10] O. Serres, V. K. Narayana, and T. A. El-Ghazawi, "An Architecture for Reconfigurable Multi-core Explorations," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEE, 2011, pp. 105–110.

[11] S. Wong, L. Carro, M. Rutzig, D. M. Matos, R. Giorgi, N. Puzovic, S. Kaxiras, M. Cintra, G. Desoli, P. Gai, S. A. Mckee, and A. Zaks, *Reconfigurable Computing - ERA – Embedded Reconfigurable Architectures*. Springer New York, 2011.

[12] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*, UG585 (v1.7) ed., Xilinx Inc., Feb. 2014.

[13] P. Kaufmann and M. Platzner, "Multi-objective Intrinsic Hardware Evolution," in *Intl. Conf. Military Applications of Programmable Logic Devices (MAPLD)*, 2006.

[14] T. Knieper, P. Kaufmann, K. Glette, M. Platzner, and J. Torresen, "Coping with Resource Fluctuations: The Run-time Reconfigurable Functional Unit Row Classifier Architecture," ser. LNCS, vol. 6274. Springer, 2010, pp. 250–261.

[15] P. Kaufmann, K. Glette, T. Gruber, M. Platzner, J. Torresen, and B. Sick, "Classification of Electromyographic Signals: Comparing Evolvable Hardware to Conventional Classifiers," *IEEE Trans. Evolutionary Computation*, vol. 17, no. 1, pp. 46–63, 2013.

[16] R. Dobai and L. Sekanina, "Towards evolvable systems based on the Xilinx Zynq platform," in *IEEE International Conference on Evolvable Systems (ICES)*. IEEE, 2013, pp. 89–95.

[17] B. Sprunt, "The Basics of Performance Monitoring Hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002.

[18] 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.

[19] 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.

[20] J. A. Walker, J. F. Miller, P. Kaufmann, and M. Platzner, "Problem Decomposition in Cartesian Genetic Programming," in *Cartesian Genetic Programming*, ser. Natural Computing Series. Springer Berlin Heidelberg, 2011, pp. 35–99.

[21] P. Kaufmann, *Adapting Hardware Systems by Means of Multi-Objective Evolution*. Berlin: Logos Verlag, 2013.

[22] 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.

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