# Adaptive Cache Infrastructure: Supporting dynamic Program Changes following dynamic Program Behavior

Fabian Nowak Rainer Buchty
Wolfgang Karl
Universität Karlsruhe (TH), Institut für Technische Informatik (ITEC)
Zirkel 2, 76131 Karlsruhe, Germany
{nowak|buchty|karl}@ira.uka.de

**Abstract:** Recent examinations of program behavior at run-time revealed distinct phases. Thus, it is evident that a framework for supporting hardware adaptation to phase behavior is needed. With the memory access behavior being most important and cache accesses being a very big subset of them, we herein propose an infrastructure for fitting cache accesses to a program's requirements for a distinct phase.

## 1 Introduction

When regarding a program's long-time behavior, it is evident that each program consists of at least three different phases: the first one can be called the *initialization phase*, the second one the *main* or *computational phase*, and the last one the *final* or *termination phase* [LY91]. It can even be shown that after millions of instructions in the so-called *main phase*, new phases of program execution commence [SPH+03]. Metrics changing from phase to phase include, but are not limited to, branch prediction, cache performance, value prediction, and address prediction [SC99, BABD03].

By providing an architecture tailored at only one phase, as is done usually, this very phase is executed with best results concerning the aspired enhancements, i.e. performance or energy efficiency. By means of reconfiguration, however, we are able to support a program during its whole run-time when adapting the hardware in every distinct phase.

In this paper, we address the basic concept, present our latest implementation results, and show in detail how much cache reconfiguration can possibly speed up program execution by giving benchmark [GRE+01] results. Furthermore, we show a simple means to handle phase changes more dynamically.

This is especially interesting for scientific super-computing where slight enhancements of the whole system can result in some fewer days of computation or less energy consumption, and therefore, cooling and money savings. Another interesting aspect is to further speed up execution of parallel computing nodes sharing some memory levels, like L2 cache and main memory, by dynamically partitioning a cache's area to the different processors' needs.

The rest of this paper is organized as follows: as a start, an overview of the finished and ongoing work is given. Upon that base, in Section 3 we present a novel architecture for supporting cache reconfiguration at reunite. The current state of our implementation is then summed up in Section 4 giving a first impression of how much speed-up can be achieved. This is explained in detail in Section 5. In order to round off the whole work, an application supporting the tool-chain is illustrated. The paper finishes with the conclusion.

## 2 Related work

Much work was done already in the vast field of cache partitioning and cache adaptation. As far as cache partitioning is concerned, dividing into instruction and data caches is a very well-known method for increasing cache hit-rates. Other methods are partitioning into scalar and array data caches [NKOF07] or separating by temporal and spatial locality [GAV95]. The last approach was extended by Johnson and Hwu by means of memory address tables yielding a speed-up of up to 15% [JmWH97]. They request a framework for intelligent run-time management of the cache hierarchy. Partitioning can also be used for offering instruction reuse buffers based on cache technology as is done by Ranganathan et al. in [RAJ00]. Unfortunately, some software changes have to be made for the system to work. Suh et al. examined dynamic partitioning of shared cache memory [SRD04] and come to the conclusion that re-partitioning only needs to take place when a context switch occurs.

Cache adaptation, which may benefit of cache partitioning, requires the micro-system to monitor its memory system performance, detect changing demands, and initiate reconfiguration of at least a subpart of the memory system. Benitez et al. evaluate performance and energy consumption of a cache system, which is very limited concerning possible parameter changes. With their micro-architecture of the Field-Programmable Cache Array (FPCA), they also introduce basic phase detection where branches are counted as a simple means to recognize possible changes related to the memory system [BMRL06]. When reconfiguring, any cache content is lost, thus reconfiguration has to be controlled with the processor initiating a cache flush in advance.

More sophisticated phase detection is achieved by Sherwood et al. based on basic block vectors and clustering [SPHC02]. Similarly, Balasubramonian and Albonesi managed phase detection by measuring branch frequencies and cache misses. Upon this information, cache parameter adjustment is carried out, such as sizes, associativities and access latencies [BABD03]. In their latest work, Huffmire and Sherwood applied wavelet-based phase detection onto L1 accesses and are able to accurately predict L2 miss rates, and thus, phases [HS06].

With their work towards reconfigurable cache memory, Ogasawara et al. proved that varying certain cache parameters indeed makes sense [OTW+06]. Despite delivering first results, their system is only tailored at simulation and very limited with respect to variety of dynamic parameters.

In the embedded field, Vahid and Gordon-Ross among others are developing configurable cache controllers with a strong focus on embedded applications and energy consumption [ZVN03, GRVD04, GRV07, GRZVD04].

A concept of the required tool-chain for a reconfigurable system was worked out by Mei et al. [MVML01]. It consists of a two-level approach that is based on profiling and mapping and should be adaptable to the hardware infrastructure under development at our chair.

# 3 Reconfigurable Cache Architecture

The first goal of our architecture was the creation of a cache capable of acting both as Level-1 and Level-2 cache. Secondly, it has to be reconfigurable. As a last aspect, we want the cache controller to be synthesizable for hardware usage.

While most reconfigurable caches depend on the reconfiguration capabilities of the underlying FPGA chip, our implementation handles the reconfiguration logically, only by hardware logic in the controller itself. This decision offers a great degree of freedom in choosing different sizes for both the cache memory and its control and replacement information, in running with different associativities and in changing replacement strategies.

The cache only caches accesses to the ordinary main memory, which consists of DDR-SDRAM on the Xilinx ML310 and ML403 evaluation boards available at our institute.

## 3.1 Cache Structure

As already mentioned, we decided to split the whole cache into three distinct areas: *cache memory*, *control memory* and *replacement memory*. The *control memory* indicates whether the according cache line is valid, modified, partly valid and additionally stores the tag. The *replacement memory* is only needed, when an associativity of more than one and a more sophisticated replacement strategy like LRU is used. In Figure 1, the layout of a 2-way set-associative cache with eight lines and a replacement strategy like FIFO, Pseudo-LRU, or LRU is illustrated.

## 3.2 Cache Controller

The cache controller handles several things: not only is it responsible for serving read/write requests to the cache memories explained above, but also for initiating the reconfiguration process, which in return is executed by the *reconfiguration controller*, and providing a *monitoring unit* for access data aggregation. The controller employs distinct buses for its individual tasks: it interfaces to the external DDR-SDRAM, to which memory read/write requests from the PLB are forwarded if no entry is found in the cache. Furthermore,

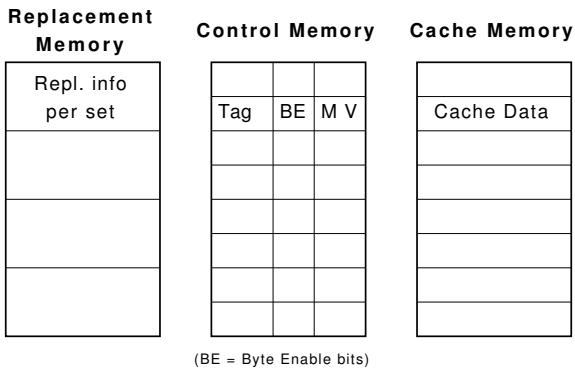


Figure 1: Exemplary cache layout

the controller features a dedicated DCR bus connection for reconfiguration purpose and monitoring control, and a dedicated monitoring interface to output monitor data.

The controller's *monitoring unit* provides therefore a 64 bit wide output register containing information about cache hits and misses, the conflict-causing line number, and additionally the complete memory address. Hence, it is possible to both collect information for on-line and off-line analysis, and achieve phase prediction such that the processor can initiate cache reconfiguration itself or have the system automatically adopt to the detected phase changes. The monitor output register is depicted in Figure 3.

As a reaction to program phases, reconfiguration might take place. Cache reconfiguration is achieved by requesting the controller to enter the reconfiguration state, then writing new parameterization values, explicitly indicating the end of the reconfiguration request, and waiting for the reconfiguration process to finish. Meanwhile, the processor could do different tasks not involving the cached main memory. But for ease of implementation and in order to avoid additional code for busy waiting, we decided to simply halt the processor. In [NBK07], we already proved that several traditionally fixed cache parameters are reconfigurable and presented some implementation approaches using heuristics where necessary. The complete process is illustrated in Figure 2.

## 4 Implementation Results

Figure 4 illustrates the complete extended cache/memory controller, including all interfaces and sub-modules controlling these interfaces or accessing them.

We are currently in process of synthesizing the whole design for the Virtex-II Pro and the Virtex-4 FX12 in order to benchmark the system in real hardware. First results show that some modifications still have to be made to the design; especially, the maximum clock rate is only at 9.779 MHz. This is due to the fact that the processor clock is also used for lots of comparisons for memory accesses dependent on the chosen associativity and number

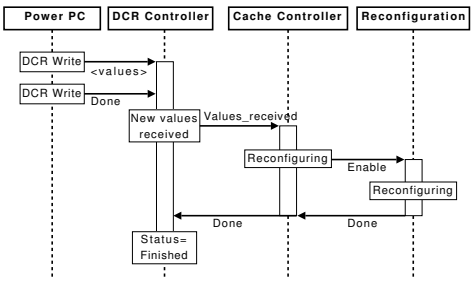


Figure 2: Reconfiguration flow

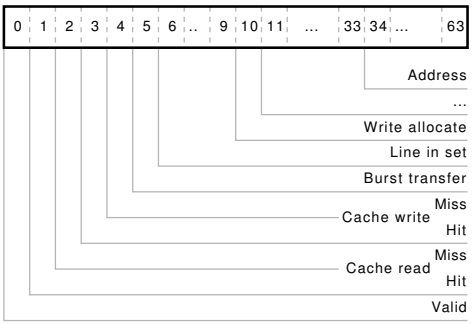


Figure 3: Monitor output register: the unused bits are intentionally reserved for further extensions to the current monitoring interface.

of sets. Hence, the core is currently undergoing re-design for a Virtex-4 FX100 to better match current FPGA's hardware resources, achieving much improved clock rates.

In Table 1, we present the hardware resource usage of parts of the current implementation targeting the Virtex-II Pro, that is to say of the cache controller, the reconfiguration module, the DCR controller, the monitoring component and the basic PLB DDR Components. We again want to emphasize that these numbers reflect the very first concept study and are not to be taken as architecture-optimized numbers for production use in computing systems.

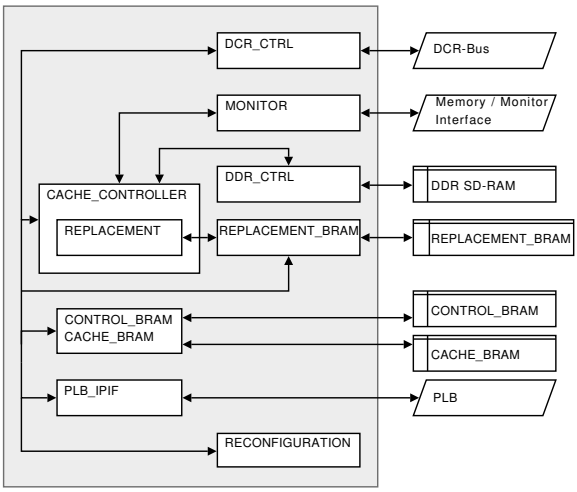


Figure 4: Reconfigurable Cache Controller Architecture

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| **Total Number Slice Registers** | 2089 | 27392 | 7.6% |
| Number used as Flip Flops | 1826 | | |
| Number used as Latches | 263 | | |
| Number of 4 input LUTs | 37980 | 27392 | 138.7% |
| **Logic Distribution** | | | |
| Number of occupied Slices | 20212 | 13696 | 147.7% |
| **Total Number of 4 input LUTs** | 38660 | 27392 | 141.1% |
| Number used as logic | 37980 | | |
| Number used as a route-thru | 680 | | |
| Number of MULT18X18s | 2 | 136 | 1.471% |
| Number of GCLKs | 10 | 16 | 62.5% |
| **Total equivalent gate count for design** | 268036 | | |
| Additional JTAG gate count for IOBs | 79488 | | |

Table 1: Hardware resource usage of synthesized components.

## 5 Application Speed-Up

Given the benchmark results from [GRE+01], we will show how much speed-up can be obtained. First, we write down the rather simple formula for calculating whether reconfiguration is appropriate and for comparing the overall memory access time with reconfiguration to the overall memory access time without adaptation to program phases.

$$\begin{aligned} & n_{wm} * t_{wm} + n_{wh} * t_{wh} + n_{rm} * t_{rm} + n_{rh} * t_{rh} && > \\ & n'_{wm} * t_{wm} + n'_{wh} * t_{wh} + n'_{rm} * t_{rm} + n'_{rh} * t_{rh} && + t_{reconfiguration} \end{aligned} \quad (1)$$

where $n_{wm}$ indicates the number of write misses, $t_{hr}$ the time required for serving a read request when a hit in the cache occurs, and $n'$ the numbers achieved after reconfiguration. As can be seen easily, the overall memory access time is the sum of the cycles needed in all program phases plus their respective reconfiguration times.

Then we want to outline the time needed for both cache accesses and reconfiguration time measured in clock cycles. Table 2 gives a comparison of the implemented memory access times with and without our cache. Obviously, most speed-up can be achieved by increasing the read hit rate, while the write accesses do not contribute too much.

| Access type | Duration w/o cache | Duration w/ cache |
|---|---|---|
| Read Hit | 15 | $8 + \lfloor n/2 \rfloor$ |
| Read Miss | 15 | $15 + \lfloor (a-1)/2 \rfloor$ |
| Write Hit | 9 | $8 + \lfloor n/2 \rfloor$ |
| Write Miss | 9 | $10 + \lfloor (a-1)/2 \rfloor$ |

$n$ denotes the line in the set where the hit occurs; $a$ denotes the level of associativity.

Table 2: Access times to main memory without and with cache (using write-through)

Already upon that basis, we are able to extend Equation 1 to the following for a one- and two-way set-associative cache:

$$\begin{aligned} c * (p_{wm} * 10 + p_{wh} * 8 + p_{rm} * 15 + p_{rh} * 8) \quad & > \\ c' * (p'_{wm} * 10 + p'_{wh} * 9 + p'_{rm} * 15 + p'_{rh} * 9) \quad & + t_{reconfiguration} \end{aligned} \tag{2}$$

where $n_\alpha = p_\alpha * c$, $c$ the number of memory access cycles "per phase", and $\sum_i p_i = 1$. Of course, we have to pay attention to use the best case access times for the first part, while in contrast the worst case has to be regarded for the second part.

Regarding reconfiguration time, it must be noted that for area and memory concerns, the reconfiguration of associativity only saves half of the cache's content. This decision makes reconfiguration up to 37% faster. The process is thus split into two phases with the first one being responsible for assuring cache consistency of the "rear half" by executing write-back where necessary and the second one moving cache lines to their new locations. The first step takes either 2 or 12 cycles per cache line, depending on whether write-back is needed. If write-through is used for write accesses, this step is only responsible for invalidating the "rear" cache lines. The second phase then takes 3 cycles per cache line for rearranging cache line data (this is indeed where reconfiguration would need double the time if the whole cache content was kept).

Hence, for doubling associativity, a one-way set-associative cache with 1024 lines and write-through strategy requires $512 * 2 + 512 * 3 = 2560$ cycles (ignoring the few setup cycles of each reconfiguration step).

Now, the memory access count stays the same when executing a disinct phase with a different configuration, thus, $c = c'$. In addition, we assume an enhancement of cache hit rate of 20% as stated in [ZVN03] when going from one-way to two-way associativity. We further assume $p_{wm} = p_{wh} = p'_{wm} = p'_{wh} = 0.25$, $p_{rh} = p_{rm} = 0.25$. With this enhancement, we get $p'_{rh} = 0.3$ and accordingly $p'_{rm} = 0.2$. Equation 2 therefore becomes

$$\begin{aligned} c * (0.25 * 10 + 0.25 * 8 + 0.25 * 15 + 0.25 * 8) \quad & > \\ c * (0.25 * 10 + 0.25 * 9 + 0.2 * 15 + 0.3 * 9) \quad & + 2560 \end{aligned} \tag{3}$$

$$\Leftrightarrow$$

$$c * (0.25 * 7 + 0.5 * 15 - 0.3 * 9) \quad > 2560$$

$$\Leftrightarrow$$

$$c > 2560/(1.75 + 7.5 - 2.7) \quad = 390.84 \tag{4}$$

Hence, after only 391 memory accesses, the reconfiguration effort proved sensible.

## 6 Toolchain Integration

In order to simplify the correct and consistent parameterization of the rather complex-to-configure cache-controller, we developed a program with a graphical user interface, which automatically adjusts all parameters with respect to the user's demand.

The *configurator* program has to be invoked manually after having designed the system-on-chip in *Xilinx Platform Studio (XPS)* [Xi07]. It then enables the user to e.g. specify a maximum associativity of 4, while in the beginning of the execution, an associativity of 2 is chosen. Furthermore, it ensures that the data widths of the replacement and control information are wide enough to store all information required. Having written the changes, the user can return to *XPS* and undertake remaining changes, synthesize the description, and download the bitstream to the device. The configuration task flow is depicted in detail in Figure 6.


Figure 5: Graphical User Interface for a-priori configuration of the cache/memory system

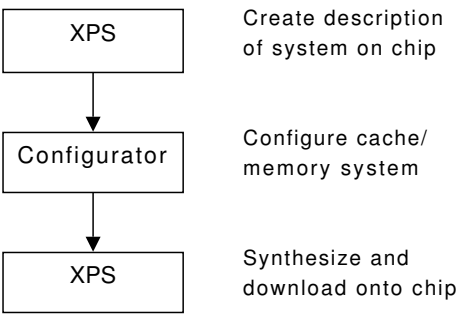


Figure 6: Task flow for initial configuration of the cache/memory system

## 7 Conclusions and Future Work

We have presented an architecture concept for supporting exploitation of program phase behavior by adapting a subset of the memory system – the cache system – to a program's needs. With such an architecture, it is possible to not only statically reconfigure the system every $n$ instructions, but to have the operating system decide itself whether it seems advantageous to use another configuration. This is where the particular strength of our approach becomes obvious: the time needed for reconfiguration can be clearly estimated between

lower and upper bounds, which do not differ too much. In contrast, when using dynamic FPGA reconfiguration, parts of the system have to be halted and due to side effects, the reconfiguration time may vary unpredictably.

Future work includes the development of a monitoring component, which is capable of indicating new phases, and operating system functions for phase and system evaluation by use of the gathered monitoring information and application-embedded hints. These hints can easily be determined manually by visualization tools like in [TK05] and inserted into binary code.

We also intend to work on faster main memory access by a more direct connection of the memory to the processor and on using wider cache-lines. Additionally, offering hardware acceleration units in the unused cache area seems interesting [KST01].

## References


[BABD03] Balasubramonian, R., Albonesi, D. H., Buyuktosunoglu, A., und Dwarkadas, S.: A dynamically tunable memory hierarchy. *IEEE Trans. Computers*. 52(10):1243–1258. 2003.

[BMRL06] Benitez, D., Moure, J. C., Rexachs, D. I., und Luque, E.: Evaluation of the field-programmable cache: performance and energy consumption. In: *CF '06: Proceedings of the 3rd conference on Computing frontiers*. S. 361–372. New York, NY, USA. 2006. ACM Press.

[GAV95] Gonzàlez, A., Aliagas, C., und Valero, M.: A data cache with multiple caching strategies tuned to different types of locality. In: *ICS '95: Proceedings of the 9th international conference on Supercomputing*. S. 338–347. New York, NY, USA. 1995. ACM Press.

[GRE+01] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., und Brown, R. B.: Mibench: A free, commercially representative embedded benchmark suite. In: *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. S. 3–14. Washington, DC, USA. 2001. IEEE Computer Society.

[GRV07] Gordon-Ross, A. und Vahid, F.: Dynamic optimization of highly configurable caches for reduced energy consumption. *Riverside ECE Faculty Candidate Colloquium*. March 2007. Invited Talk.

[GRVD04] Gordon-Ross, A., Vahid, F., und Dutt, N.: Automatic tuning of two-level caches to embedded applications. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. S. 10208. Washington, DC, USA. 2004. IEEE Computer Society.

[GRZVD04] Gordon-Ross, A., Zhang, C., Vahid, F., und Dutt, N.: Tuning caches to applications for low-energy embedded systems. In: Macii, E. (Hrsg.), *Ultra Low-Power Electronics and Design*. Kluwer Academic Publishing. June 2004.

[HS06] Huffmire, T. und Sherwood, T.: Wavelet-based phase classification. In: *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. S. 95–104. New York, NY, USA. 2006. ACM Press.

[JmWH97] Johnson, T. L. und mei W. Hwu, W.: Run-time adaptive cache hierarchy management via reference analysis. In: *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*. S. 315–326. New York, NY, USA. 1997. ACM Press.

[KST01] Kim, H., Somani, A. K., und Tyagi, A.: A reconfigurable multifunction computing cache architecture. *IEEE Trans. Very Large Scale Integr. Syst.* 9(4):509–523. 2001.

[LY91] Lim, H.-B. und Yew, P.-C.: Parallel program behavioral study on a shared-memory multiprocessor. In: *ICS '91: Proceedings of the 5th international conference on Supercomputing*. S. 386–395. New York, NY, USA. 1991. ACM Press.

[MVML01] Mei, B., Vernalde, S., Man, H. D., und Lauwereins, R. Design and optimization of dynamically reconfigurable embedded systems. 2001. `http://citeseer.ist.psu.edu/mei01design.html`.

[NBK07] Nowak, F., Buchty, R., und Karl, W.: A run-time reconfigurable cache architecture. In: *Proceedings of the 2007 Parallel Computing Conference, Aachen/Jülich*. September 2007.

[NKOF07] Naz, A., Kavi, K., Oh, J., und Foglia, P.: Reconfigurable split data caches: a novel scheme for embedded systems. In: *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. S. 707–712. New York, NY, USA. 2007. ACM Press.

[OTW+06] Ogasawara, Y., Tate, I., Watanabe, S., Sato, M., Sasada, K., Uchikura, K., Asano, K., Namiki, M., und Nakajo, H.: Towards reconfigurable cache memory for a multithreaded processor. In: Arabnia, H. R. (Hrsg.), *PDPTA*. S. 916–924. CSREA Press. 2006.

[RAJ00] Ranganathan, P., Adve, S., und Jouppi, N. P.: Reconfigurable caches and their application to media processing. In: *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. S. 214–224. New York, NY, USA. 2000. ACM Press.

[SC99] Sherwood, T. und Calder, B. The time varying behavior of programs. August 1999. Technical Report UCSD-CS99-630, University of California, San Diego.

[SPH+03] Sherwood, T., Perelman, E., Hamerly, G., Sair, S., und Calder, B.: Discovering and exploiting program phases. *IEEE Micro*. 23(6):84–93. 2003.

[SPHC02] Sherwood, T., Perelman, E., Hamerly, G., und Calder, B.: Automatically characterizing large scale program behavior. In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. S. 45–57. New York, NY, USA. 2002. ACM Press.

[SRD04] Suh, G. E., Rudolph, L., und Devadas, S.: Dynamic partitioning of shared cache memory. *J. Supercomput.* 28(1):7–26. 2004.

[TK05] Tao, J. und Karl, W.: Optimization-oriented visualization of cache access behavior. In: *Proceedings of the 2005 International Conference on Computational Behavior (Lecture Notes in Computer Science 3515)*. S. 174–181. Springer. May 2005.

[Xi07] Xilinx, Inc. Platform Studio and EDK Details. 2007. Web site: `http://www.xilinx.com/ise/embedded/edk_pstudio.htm`.

[ZVN03] Zhang, C., Vahid, F., und Najjar, W.: A highly configurable cache architecture for embedded systems. In: *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*. S. 136–146. New York, NY, USA. 2003. ACM Press.