Adaptive Cache Structures

Carsten Tradowsky(✉), Enrique Cordero, Christoph Orsinger, Malte Vesper, and Jürgen Becker

Institute for Information Processing Technologies, Karlsruhe Institute of Technology, Karlsruhe, Germany
{tradowsky,becker}@kit.edu,
{enrique.cordero,christoph.orsinger,malte.vesper}@student.kit.edu

Abstract. Novel programming paradigms enable the concurrent execution and the dynamic run-time rescheduling of several competing applications on large heterogeneous multi-core systems. However, today the cache memory is still statically allocated at design time. This leads to a distribution of memory resources that is optimized for an average use case. This paper introduces adaptive cache structures to be able to cope with the agility of dynamic run-time systems on future heterogeneous multi-core platforms. To go beyond the state of the art, the cache model is an implemented HDL realization capable of dynamic run-time adaptations of various cache strategies, parameters and settings. Different design trade-offs are weighted against each other and a modular implementation is presented. This hardware representation makes it possible to deeply integrate the adaptive cache into an existing processor microarchitecture. The contribution of this paper is the application-specific run-time adaptation of the adaptive cache architecture that directly represents the available memory resources of the underlying hardware. The evaluation shows very efficient resource utilization while the cache set size is in- or decreased. Also, performance gains in terms of cache's miss rate and application's run-time are shown. The architecture's capabilities of performing in a multi-core use case and the potential for future power savings are also presented in an application scenario.

1 Introduction

Today's performance gains are mainly powered by a strict application of principles. Frequency gains are achieved by miniaturization of technology and making excessive use of parallelism through multi-core architectures. Since the possibilities for improvement are limited, it is time to turn to abstraction and generalization. Every abstraction or generalization comes at a cost in terms of performance or accuracy and hence has to be carefully weighted. It is necessary to take this one step further to unleash another quantum of performance from a multi-core system.

Performance gains by increasing frequency have hit the power-wall and the field has turned towards parallelism [6]. Today's multi-core systems exploit many heterogeneous resources that are allocated dynamically during run-time to different tasks [7]. The number of resources available on a chip can be varied with such adaptive architectures [8]. The benefits of reducing abstraction and giving

F. Hannig et al. (Eds.): ARCS 2016, LNCS 9637, pp. 87–99, 2016.
DOI: 10.1007/978-3-319-30695-7_7

the programmer control over hardware parameters is studied. This control allows the programmer to adapt the hardware to the application's needs. This paper presents this approach for adaptive caches in particular.

The cache design is a trade-off between power and performance, for which there is no general optimal solution [3]. However, there are optimal solutions for specific programs, in terms of power, performance, or performance per Watt [2]. One solution that is suggested by research projects like 'invasive computing' [7] is to relieve the chip designer of this burden and allow the application developer to unleash higher performance. Consequently, the application developer should choose the optimal cache configuration [4]. Further choices could be made either by the compiler or run-time system according to the current system state. This choice will be put beyond the silicon implementation stage to exploit performance gains that are higher than the cost arising from added complexity. While this strips an abstraction layer, which makes the cache truly transparent, it is a necessary step in uncovering additional performance. In parallel computing, caches are seen as something, for which cache oblivious algorithms could be written [5].

The paper is structured as follows: Sect. 2 then details the design of the adaptive cache structures. Afterwards, the adaptation possibilities are presented in Sect. 3. To prove our concept, we provide an evaluation of our adaptive cache structures in Sect. 4 Concluding, we summarize our contribution and provide an outlook for future work.

2 Designing Adaptive Cache Structures

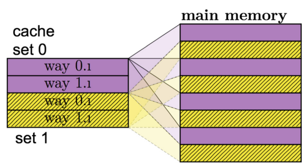
Before explaining the actual concept, a summary of the requirements will be presented to justify and explain the following concept. The ultimate goal of this design is to target a silicon implementation as an Application-Specific Integrated Circuit (*ASIC*). Currently for prototyping and development purposes, the design has been implemented using the *Leon3* on Xilinx Virtex-5 Field Programmable Gate Arrays (*FPGA*). To allow future *ASIC* implementation, the proposed design run without any *FPGA*-specific techniques like run-time partial reconfiguration. Instead, the concept is based on run-time resource reallocation.

2.1 Basic Cache Model

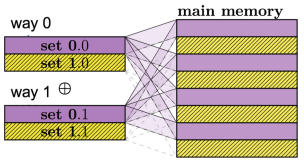
Since caches are smaller than the memory cached by them, a mapping is required. Therefore, the cache is split in so called *lines.* A line is the smallest consecutive part that can be mapped. In the simplest case, referred to as *direct mapped* cache, the main memory is split into equal size regions and each region is mapped to one line. A group of lines is called a set. To loosen the restriction of direct mapped caches, *associativity* has been introduced. In associative caches multiple cache lines are mapped to the same set-bits. For this purpose a power of two number of lines are grouped to a *set.* The number of lines in a set is called *set size* s. The memory chunks are now mapped to the sets. Therefore, there are s possible lines, in which the data can be cached. By increasing the associativity and keeping the cache size the same, the number of sets $\#s$ is decreased.

For the concept, which we present in this paper, it is favorable to reinterpret the memory mapping. Figure 1 shows the memory mapping for a two-way cache. In Fig. 1a the two-way cache is shown as one large memory. This interpretation is favorable to explain the addressing of the cache memory. Figure 1b interprets a two-way cache as two parallel caches. This reinterpretation of the cache is favorable for our concept as it enables us to directly represent the underlying hardware memory resources.

Furthermore, when using the second interpretation, it is important that any value can only be cached in one of the two caches (ways), but for retrieval the value is looked up in both caches.



(a) Set partitioning



(b) Way partitioning

Fig. 1. Mapping for two-way cache - Memory mappings for a two-way associative cache. Figure a groups the cache by sets, while Fig. b groups the cache by way.

2.2 Modularization

To meet the requirements, the cache is divided into modules as shown in Fig. 2. The modules are: 1. the cache controller, 2. the cache memory, 3. the memory controller, 4. the transceiver, and 5. the adaptation controller.

Figure 2 also shows the minimal necessary connections between the different modules needed in order to exploit the parallelization capabilities of the cache subsystem to the fullest. Half-duplex connections support either read or write transfers, while full duplex connections support both at the same time. While the *CPU*-connection type is dictated by the *CPU*, and the connection between transceiver and memory controller is implicitly given by the bus capabilities, all internal connections can be adapted as needed. In the following sections the individual modules are presented.

2.3 Cache Controller

The cache controller is the heart of the system. In addition to the standard states, it has been expanded to include an adaptation state to change the configurations. All basic states are extended to account for the adaptation as well. From the *idle* state there are five states branching out: 1. read, 2. write, 3. flush, 4. discard, and 5. adapt. The added adaptation sequence will later be explained in more detail.

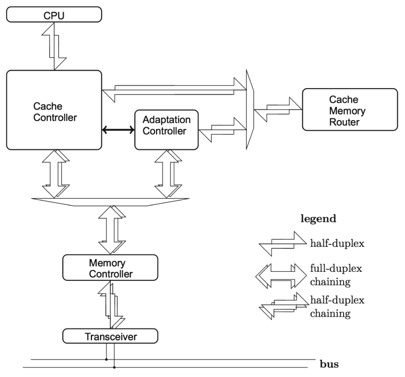
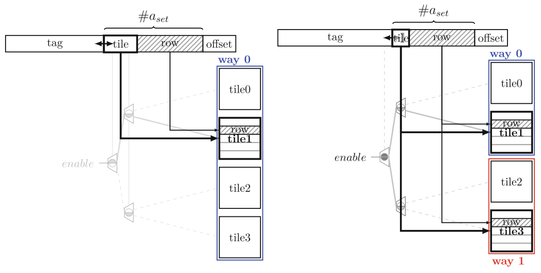


Fig. 2. Overview of the modules - Modules composing the cache system and their connections.

2.4 Cache Memory

The cache memory consists of two main parts: the tile cluster and the individual tiles.

Tile Cluster. The tile concept comes from looking at caches with associativity n as n parallel caches. Any memory location is cached only once, however, the cache is chosen by a replacement strategy. If we use additional bits from the address instead of a replacement strategy to select the cache, it behaves like a large direct mapped cache. These n caches are called *tiles*.



(a) direct mapped configuration (b) two-way configuration

Fig. 3. Tile network - The black arrows show the schematic address-mapping, the grayed out multiplier/multiplexer network shows the implementation.

If the associativity is reduced, there is no need to query all tiles, therefore a reduction in the power consumption can be achieved. The connection of multiple tiles form the actual memory of the cache.

The set address a_{set} is split into two parts like shown in Fig. 3. A static part, the *row address* a_{row} that identifies the row inside a tile, and a dynamic part, the *tile address* a_{tile}, that identifies the tile in the way. The tree of multiplexers are controlled by the a_{tile} bits of the address as shown in Fig. 3. The multiplexer on the highest level of the binary tree is controlled by the most significant bit of a_{tile}. If the associativity is increased, meaning several stages of the tree act as multipliers rather than multiplexers, the first level acting as multiplexer is controlled by the most significant bit of a_{tile}. Since increasing the associativity by n reduces the $\#a_{tile}$ ($\#a_{set}$) by n the mapping of the stages to the address bits remains the same, regardless of the currently set associativity. This becomes clear when comparing Fig. 3a with Fig. 3b. Note that all tiles are connected by an internal bus relaying the tag and row, however, only those that are *enabled* check for hits and output data on the data bus.

Tile. Three independent memory blocks are used to store the data, the tag and the control bits.

The tag *RAM* and data *RAM* hold tag and data respectively. The control bit *RAM* holds valid and dirty bits plus any per-line information needed by the replacement strategy. The data is split to avoid unnecessary read-/write-accesses.

The separation in these three RAM blocks has a potential on reducing power consumption because it avoids unnecessary read operations when only part of a memory line needs to be updated. When updating a part of the line only the data and the control bits (which are located in separate RAMs) need to be updated and while the tag stays unchanged. This reduces the amount of write operations because the tag will not be unnecessarily rewritten.

If power saving is a larger issue than performance, the access to the different *RAM* cells can be serialized. For example instead of reading the tag, control bits and data at the same time and having logic decide whether the data is put onto the data bus of the tile cluster one could first match the tag and compare the valid bit. The data RAM will only be accessed if it actually contains the value to be fetched. This costs an additional cycle, however there will only be one instead of s read operations executed on the data RAM.

3 Adaptation

The system supports different adaptation possibilities. The replacement strategy can be selected on the fly and cache memory parameters can be changed during run-time.

3.1 Configuration Register

All configuration information is stored in the *configuration register* located in the cache controller. A write to this register triggers an adaptation. Reading the register yields the current configuration.

The cache configuration register composition is described in the following paragraph but the actual sizes of the different components are omitted since they depend on the parametrization of the cache. To ease software development one should define fixed sizes independent of the parametrization.

The configuration register and its contents are shown in Fig. 4. Only powers of two values are accepted as line lengths, because otherwise irregularities are introduced in the memory mapping.

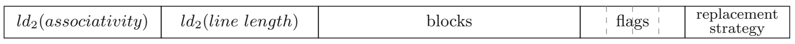


Fig. 4. Adaptive cache configuration register

3.2 Cache Memory

There are three parameters that can be used to adapt the cache memory: 1. associativity s, 2. size $\#l^a$, and 3. line length L_W.

Using these parameters, all other configuration values for the cache can be computed. Adaptation of cache memory is broken down into base cases. Every base case has two subcases, increasing or decreasing one of the aforementioned cache memory parameters. Figure 5 gives an overview of the adaptation paths and the implied costs. The following subsections detail the different adaptive cache configurations and explain the cost of adaptively changing the cache configuration. It should be emphasized that all these possibilities do not require a full cache flush.

Associativity. Associativity is quantized by the tile-concept. While the direct mapped configuration offers only one valid location, the two-way associative configuration has two. This implies that by changing the associativity, values might end up in invalid locations.

Reducing associativity requires sorting. An associativity of 2^n has 2^n lines that are allowed to store values from a set address. If the associativity is decreased from 2^n to 2^m then 2^{n-m} cells might hold values which they are not allowed to hold. Due to the mapping chosen, the 2^m new lines are a subset of the 2^n lines. Therefore, those lines can remain untouched. The valid locations of any address a for a lower associativity form a true subset of those for a higher associativity. Thus, decreasing the associativity can yield wrongly placed data regarding the new associativity.

Increasing associativity does not require sorting. As shown under the decrease of associativity, the valid locations for higher associativities form a superset for those of lower associativities. Thus, all data remains valid and no scrubbing is required.

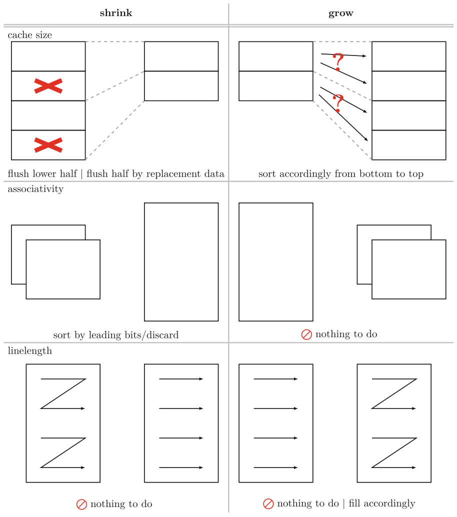

shrink
grow
cache size
?
?
flush lower half | flush half by replacement data
sort accordingly from bottom to top
associativity
sort by leading bits/discard
⊘ nothing to do
linelength
⊘ nothing to do
⊘ nothing to do | fill accordingly

Fig. 5. Overview of the basic adaptation cases. Combining these spans the entire configuration space.

Cache Size. Like associativity, we quantize cache size based on the size of a tile. The reason behind this is that tiles can be powered off to conserve energy or be reused as a whole. To make changes in cache size perpendicular to changes in associativity we assume that they are achieved by reducing/increasing the number of lines in a direct mapped cache. This concept can be applied to an s-way associative cache by looking at it as s parallel caches as explained in Subsect. 2.1. There are two possibilities to reduce the cache size based on this model: either all n-caches are reduced in size by an equal amount or m-ways are shut down. Increasing the cache size works by adding ways or increasing the set number. To increase the granularity a number of ways unequal to a power of two are allowed [1]. To achieve this the ways with the highest order identifier are shut down first. The corresponding hit, valid, and dirty signals are tied to 0.

Reducing ways requires different procedures depending on the cache configuration and on the further utilization of the reduced part. When operating in write-through mode, no further action is needed because of redundant data copies on lower cache hierarchy levels. For other operation modes, the tiles need to be flushed only if the stored values in the tile end will end up in invalid locations, or if they will be powered off.

Increasing/Reducing sets can be modelled as an increase or reduction of ways with a subsequent change in associativity. The same conditions explained in the previous sections apply for this cases.

Line Length. The line length variation works by constructing so called *virtual lines*. This means that for longer lines (multiples of the *base line length*) the line length can be simulated by the cache controller. By loading more than one line on every cache miss, the virtual line size can be increased. In order to avoid parts of the virtual line being spread over multiple sets, the way for each virtual line is determined once and then used as default for the next parts. This additional checking causes a slight overhead that can be hidden, if the tiles have a sufficient number of ports to enable checking for presence/free ways while other ways are written.

4 Evaluation of Adaptive Cache Structures

Different adaptive cache features have been presented that allow the variation of cache parameters depending on the application's current needs during run-time. This has been done with the goal of exploiting the faster run-time potential providing an optimized working point in each application by providing a more efficient individual cache setting at a given time.

As an evaluation of the proposed architecture, three main aspects will be considered. First, we will evaluate the amount of occupied hardware resources used by the adaptive cache design as well as the distribution of resources between all tiles and the tile controller based on three specific scenarios. Second, we will evaluate the impact of the adaptive cache structure on the application's and cache's performance. Third, the benefit of the adaptive cache structure in a multi-core case will be evaluated in terms of application performance.

4.1 Hardware Resources Evaluation

For utilization analysis we consider three different hardware configurations where the cache always contains 8 tiles. In each scenario the tile size is doubled with respect to the previous one, so that each design has 256, 512 and 1024 bytes per tile respectively.

Figure 6 shows the amount of reconfigurable hardware slices used for each case. It can be noted that the amount of resources used by the tile controller in each case is considerably lower (about 20 %) than the one of the tiles. In fact, considering that each scenario doubles the size of the tiles, the size of the tile controller can be considered to be almost constant with a

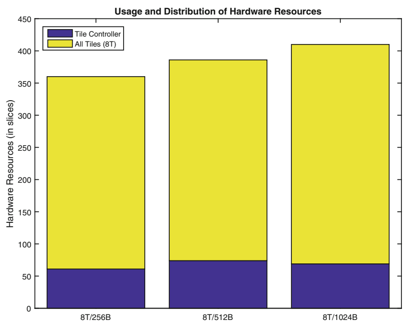


Fig. 6. Usage and distribution of hardware resources for tile controller and all tiles in three different scenarios: 256, 512 and 1024 bytes per tile with 8 tiles each.

small variation of ±10 slices. When looking at the total amount of tiles for each scenario we see increases of 4 % between the first and second scenario, and 9 % between the second and third scenario respectively.

These results show that the resources used on the FPGA for control and logic computations of the adaptive structure are comparably low considering the gain provided by each jump to the next scenario. The slices shown in Fig. 6 include all the resources used to control the cache. The actual cache memory is implemented using BRAMs. Even though the memory is being doubled in each scenario the BRAM utilization is constant for all three scenarios at 3 BRAMs per tile, thus 24 BRAMs in total. Despite the fact that the Virtex-5 XUPV5 contains BRAMs with sizes of 18 kB and 36 kB, each tile is using three separate blocks and the total amount is constant also for larger caches. This behavior can be explained by the high degree of parallel operations that need to be performed in an adaptive cache architecture. Since the BRAM blocks available are dual-port memory blocks, mapping an entire cache or an entire tile into the same BRAM would cause only an availability of two parallel access points into the memory. In the cache, however, more access points are needed in order to check in parallel all the tags depending on the associativity of the cache. This leads to an implementation of the cache in multiple blocks to allow this behavior.

4.2 Performance Evaluation

An adaptive cache reallocation architecture has the goal to provide performance gains for applications. In order to evaluate this gains we use the same three hardware scenarios explained before (256, 512 and 1024 bytes per tile).

These three scenarios will be analyzed in two different evaluations with focus on the cache performance improvements (cache miss rate) and the application's performance (run-time).

The software application running on each scenario is a Coremark benchmark. According to the benchmark's documentation a minimum of 10 s of run-time is necessary in order for the benchmark to produce meaningful results. We use a Coremark with 2000 iterations which in this architecture leads to a run-time of more than 35 s. The following tests are run on the real hardware implementation of the adaptive cache reallocation architecture on the Xilinx XUPV5.

Cache's Performance. For the analysis of the cache miss rate in this architecture, we perform two main tests: For the first test the software application is run 5 times on the previously defined hardware scenarios. Each time the application runs, a different cache configuration will be applied by the software. The configuration parameter to be changed is the amount of exclusive tiles available as cache memory for the processor. By changing the amount of tiles that the cache can use not only the cache's size will change depending on the size of each tile, but also the cache associativity will change. For each scenario we vary the amount of exclusive tiles from 4 to 8 in steps of 1 and we measure cache's miss rate during the application's run-time as a performance metric.

Figure 7 upper graph shows the behavior of the cache misses over the total cache size. We can notice a non-linear behavior and a performance gain that can be achieved when running the benchmark with a different cache configurations. For instance, a cache size of 3.5 kB built using 7 tiles of 512 bytes per tile would lead in this case to the lowest miss rate, hence the best cache performance.

In this test we selected always the same amount of tiles for each hardware scenario, which lead to different total cache sizes for each case. This is the case because each scenario has tiles of different sizes.

For the second test the same three hardware scenarios previously presented are being used. However, this time the total cache size will be varied from 2 to 6 kB in steps of 1 kB by adjusting the amount of tiles respectively for each hardware scenario. This has the advantage that the effect of the tile size and amount of tiles can be directly seen for all scenarios for each cache size. The hardware designs have been expanded to contain more available tiles as needed, however the amount of tiles to be used in each case is still configured on run-time by the software and varies depending on the hardware scenario. For example, to build a 5 kB cache in all three scenarios we would need 20 tiles of 256 bytes, 10 tiles of 512 bytes but only 5 tiles of 1024 bytes.

Figure 7 lower graph shows the different miss rates measured for each cache configuration on each hardware scenario. We see the miss rates staying in the range of 5 % to 8 % with variations in the miss rate in a range of about 2 %. The data shows that for this particular application a minimum miss rate can be achieved at a total cache size of 3.5 kB with 7 tiles of 512 bytes per tile. The same analysis for other applications would provide different results, thus showing the importance of an adaptive cache structure that can match the individual application needs and that can be controlled by the software.

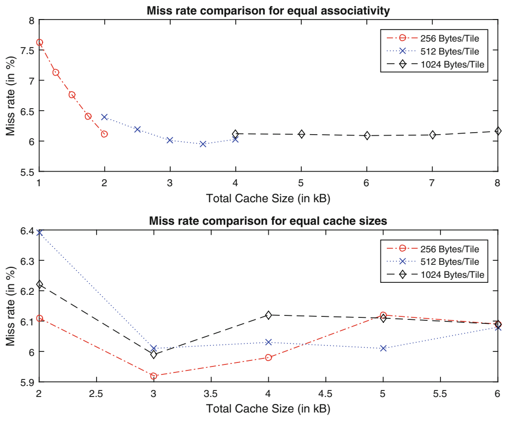


Fig. 7. Miss rate for the Coremark benchmark with 2000 iterations on three scenarios of the adaptive cache architecture with same associativity (upper graph) and with same cache size (lower graph).

This analysis is intended as an example to show the impact that the different adaptive parameters in the dynamic cache reallocation architecture can have on each individual application. Giving the developer the chance to adapt the underlying hardware characteristics to the current application, can allow the software to profit from a more suited architecture for a better performance.

Application's Performance. This second evaluation will investigate the effect of the adaptive architecture on the application's run-time of the same two tests previously described in the first evaluation. The software running in each of these tests is also the Coremark benchmark with 2000 iterations. First we consider the first test where in all three scenarios the amount of tiles is varied between 4 and 8 by the software.

Figure 8 upper graph shows the run-time in seconds over the total cache size for the three scenarios. The graph shows a strong improvement of the application run-time on small cache sizes and a much smaller variation for larger cache sizes. The graph also suggests that the performance can vary significantly with a constant cache size but with a different tile distribution or different bytes per tile. This can be seen more clearly at 2 kB cache size, where fewer tiles with 512 bytes per tile lead to a longer execution time than more tiles with 256 bytes per tile each.

Further, we consider the second test where the total cache size is varied equally among the three scenarios by adjusting the amount of tiles respectively. Figure 8 lower graph shows the results for the amount of computation cycles used for each scenario with equal cache sizes. The results show a noticeable difference for small sized caches however much lower variation in the amount of computation cycles needed as the cache gets bigger.

The results of these two evaluations for the cache's and application's performance show that by giving the application developer control of the cache configuration a performance gain can be achieved with the flexibility of this adaptive cache architecture.

It is not only important how much improvement the architecture can bring to the application, but also the potential of this architecture in a multi-core environment to provide benefits across multiple CPUs constantly increasing or reducing their momentary cache needs. This topic is more important than ever as every L1 cache miss has to go through a local bus to a 2nd level cache. If there is a miss in the 2nd level cache, which is most likely, the comparably slow DDR memory has to be accessed via the NoC, which costs a lot of time in shared memory multi-core scenarios. This aspects are considered in the following section.

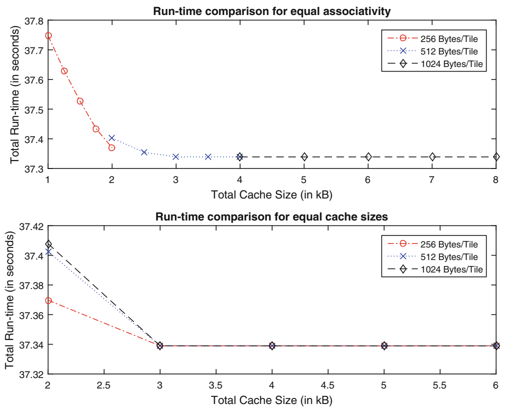


Fig. 8. Run-time for the Coremark benchmark with 2000 iterations on three scenarios of the adaptive cache architecture with same associativity (upper graph) and same cache size (lower graph).

4.3 Potential of the Adaptive Cache Architecture in a Multi-core Scenario

Reconfiguring the architecture to reallocate cache tiles with different parameters during run-time adds a reallocation overhead to the scenario. In order to evaluate the overhead and analyze the potential gain we simulate a dual-core system with each core featuring the adaptive cache architecture. The system parameters are provided in Table 1. We choose two applications, which have a different cache behaviour. The Coremark benchmark profits from more cache memory, while the MiBench (ADPCM) streaming Benchmark is not impacted by less cache memory.

Table 1. Test system configuration parameters.

Parameter	Value
Amount CPUs	2
Clock frequency	80 MHz
I-Cache size	8 kB
I-Cache associativity	1
D-Cache amount tiles	8
D-Cache tile size	256 B, 512 B
D-Cache line length	32 Bit

The test scenario consists of two stages as shown in Fig. 9. In the first stage the Coremark benchmark runs in both CPUs simultaneously with the same cache resources. In the second stage the MiBench (ADPCM) benchmark is run in CPU1, while the Coremark is run again in CPU2 with additionally reallocated cache resources from CPU1. In order to avoid false results because of cache influences, the Coremark Benchmark exists in two separate copies in the memory, such that the CPU2 uses two different copies for both stages of the test.

We compare a static configuration design with the adaptive system containing an adaption sequence between both benchmarks. Test results have shown that for the first stage of the test (Coremark benchmark running on both CPUs) with 8 tiles, a distribution of 4:4 between both cores offers the best results. Similarly, for the second stage of the test (MiBench (ADPCM) running in CPU1 and Coremark running in CPU2) a distribution of 2:6 provides the best results. The adaptive design will reallocate the resources between both test stages to compare the total run-time with a static design. The goal is to evaluate if the costs for the adaption sequence can achieve a shorter run-time because of a better distribution of resources. The static configuration will run with constant four tiles per CPU.

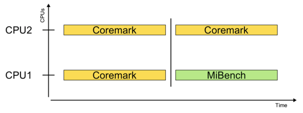


Fig. 9. Time flow of benchmarks in dual-core application scenario

Table 2 shows an overview of the results for the second stage of the test. The first row presents the static case with an equal tile distribution across both CPUs and no adaption sequence. The second row shows the results for the reallocated resources with the corresponding times for the adaption sequence. We notice a slight increase in the

Table 2. Run-time of the applications of second test stage benchmarks.

	Benchmark	Tiles	Adaption time	Run-time
CPU 1	MiBench	4	0 ns	13,31 ms
CPU 2	Coremark	4	0 ns	11,60 ms
CPU 1	MiBench	2	75 ns	13,38 ms
CPU 2	Coremark	6	880 ns	11,27 ms

run-time for CPU1 and a larger decrease in the run-time for CPU2 for the adaptive design.

An overview of the absolute differences between the static and adaptive cases can be seen in Table 3. As expected the MiBench runs slower on CPU1 however the Coremark is accelerated. It is important to note that the 853 ns of reallocation time is already included in the total run-time. Overall, we see a 2.8 % improvement after the resources reallocation.

Table 3. Relative run-time and performance gain of the dynamic cache reallocation.

	Benchmark	relative Run-time	Performance gain
CPU 1	MiBench	$+67\,\mu s$	$-0.5\,\%$
CPU 2	Coremark	$-329\,\mu s$	$+2.8\,\%$

5 Conclusion and Future Work

We provide a dynamic cache architecture, that for the first time has been described in a hardware description language and implemented on a Virtex-5 XUPV5. Our cache architecture exploits fine grain run-time adaption, which enables performance gains while keeping the hardware implementation overhead to a minimum. The presented concept provides the capability to adapt to different cache parameters on run-time and to redistribute vacant memory to other parallel processors. As the evaluation has shown, the architecture provides multiple advantages and performance gains with an expandable potential for multi-core architectures. It was shown that the hardware overhead introduced in the resource utilization of the adaptive architecture is small and slowly increasing while doubling the cache size. The evaluation also shows performance gains in both, the cache's miss rate and the application's run-time. Miss rate improvements and run-time reductions can be achieved by selecting an appropriate cache configuration through the software accessible cache configuration register. At last, the potential of the adaptive architecture in a multi-core scenario was shown by simulating a dual-core use case with adaptive cache architectures and running two different benchmarks in two CPUs. The results showed a 2.8 % improvement compared to the static case, as well as the feasibility of the adaptive architecture to reallocate tiles in a multi-core environment. Overall, this evaluation shows the importance of allowing the application developer to control hardware parameters of the underlying architecture to boost the applications performance.

For future work, the adaptive architecture will be expanded to a larger multi-core scenario running on the FPGA hardware, and tested using well established benchmarks for performance evaluations. Further, we will evaluate the power consumption as an important criterion for design decisions. Such an analysis would furthermore help to decide on which granularity cache size and associativity adjustments are sensible.

Acknowledgment. This research work is supported by the German Research Foundation (DFG) within the Transregio SFB *Invasive Computing* (DFG SFB/TRR89).

References

1. Albonesi, D.: Selective cache ways: on-demand cache resource allocation. In: MICRO-32, Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 248–259 (1999)
2. Gordon-Ross, A., Lau, J., Calder, B.: Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy. In: Proceedings of the 18th ACM Great Lakes Symposium on VLSI - GLSVLSI 2008, pp. 379–382 (2008)
3. Malik, A., Moyer, B., Cermak, D.: A low power unified cache architecture providing power and performance flexibility (poster session). In: Proceedings of the International Symposium on Low Power Electronics and Design - ISLPED 2000, pp. 241–243 (2000)
4. Nowak, F., Buchty, R., Karl, W.: A run-time reconfigurable cache architecture. Adv. Parallel Comput. **15**, 757–766 (2008)
5. Prokop, H.: Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology (1999)
6. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobb's J. **30**, 202–210 (2005)
7. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: an overview. In: Hübner, M., Becker, J. (eds.) Multiprocessor System-on-Chip, pp. 241–268. Springer, New York (2011)
8. Tradowsky, C., Thoma, F., Hubner, M., Becker, J.: Lisparc: using an architecture description language approach for modelling an adaptive processor microarchitecture. In: 7th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 279–282 (2012)