

## Performance Advantage of Reconfigurable Cache Design on Multicore Processor Systems

Jie Tao · Marcel Kunze · Fabian Nowak ·  
Rainer Buchty · Wolfgang Karl

Received: 31 October 2007 / Accepted: 3 April 2008 / Published online: 24 April 2008  
© Springer Science+Business Media, LLC 2008

**Abstract** With the trends of microprocessor design towards multicore, cache performance becomes more important because an off-chip access would be increasingly expensive due to the competition across the processor cores. A question arises: How to design the cache architecture to prevent a performance bottleneck caused by data accesses? This work studies a reconfigurable cache architecture that can be dynamically configured for meeting the individual demand of running applications. Using a self-developed cache simulator, we first examined how different cache organization and configuration influence the parallel execution of OpenMP applications. The experimental results show that applications benefit from a flexible cache with reconfigurability. This motivated us to go a step further and develop a hardware prototype of this novel architecture.

**Keywords** Cache performance · Multicore processor · Simulation · Reconfigurable architecture

---

This work was conducted as Dr. Tao worked at the Institut für Technische Informatik, Universität Karlsruhe.

---

J. Tao  
Department of Computer Science and Technology, Jilin University, Changchun, Jilin,  
People's Republic of China

J. Tao (✉) · M. Kunze  
Steinbuch Centre for Computing, Forschungszentrum Karlsruhe, Karlsruhe Institute of Technology,  
Karlsruhe, Germany  
e-mail: jie.tao@iwr.fzk.de

F. Nowak · R. Buchty · W. Karl  
Institut für Technische Informatik, Universität Karlsruhe, Karlsruhe Institute of Technology,  
Karlsruhe, Germany

## 1 Motivation

The architecture of multiprocessor systems develops in two directions: clusters with explicit message passing communication and shared memory machines which allow the processors to communicate using traditional memory loads and stores. Correspondingly, two standard programming models exist.

OpenMP [1] is a portable model supporting parallel programming on multiprocessor systems with a shared memory. It is gaining broader application in both research and industry. From high performance computing to consumer electronics, OpenMP has been established as an appropriate thread-level programming paradigm. In comparison to MPI [2], another dominating programming model for parallelization, OpenMP has several advantages. For example, MPI programmers have to explicitly send messages across processes, however, OpenMP provides implicit thread communication over the shared memory. Another feature of OpenMP is that it maintains the semantic similarity between the sequential and parallel versions of a program. This enables an easy development of parallel applications without the requirement of specific knowledge in parallel programming. Due to these features, OpenMP is regarded as a suitable model for the next generation of microprocessors.

According to Moore's law, billions of transistors can be integrated on a single chip in the near future. However, due to technical limitations the performance of conventional uniprocessors can not be improved drastically. In this case, multicore occurs as a solution. By distributing the computation across several processors, high performance could be expected.

Nevertheless, this high performance is based on an assumption that the main memory does not stall the processors for acquiring the requested data. This assumption, however, can usually not be held in the real world. First, on multicore architectures the per-processor cache is smaller in contrast to those provided by uniprocessor or SMP machines. This means more cache misses can be caused on multicores. According to an existing research work [3], up to a 4-fold of cache miss rate could be measured on a 4-processor multicore as on a uniprocessor system. Second, several cores share the same memory and as a consequence an access to the main memory can take longer due to potential bus contention. These facts indicate that on multicore processors more cache misses can be produced and an off-chip memory reference would be more expensive.

In this case, cache optimization becomes especially important for a multicore architecture to fully present its computing capacity. Such optimization was usually performed by adapting the application structure to the existing cache architecture [4,5]. On the other hand, the evolution in hardware support makes it possible to build reconfigurable components that can adapt themselves to the runtime system. This gives us the prerequisite to explore a new challenging: designing adaptive caches to fit the requirement of individual applications.

Actually, researchers have started investigations in this area with a current focus on the organization of the second level cache [6,7] and basic mechanisms for reconfigurable cache architecture [8,9]. Our goal is to implement the hardware. But before this, it is necessary to know whether applications benefit from this novel design.

To acquire the answer, we built a cache model that simulates the whole cache hierarchy of a multicore processor system, with arbitrary level of caches, different replacement policies, a variety of prefetching schemes, and different cache line invalidation strategies. Configuration parameters, like cache size, set size, block size, and associativity, can be specified by users. Based on this simulator, we primarily studied the influence of private and shared L2 cache to the performance of OpenMP applications and the benefit of cache reconfiguration.

The simulation results motivated us to further design and develop a hardware prototype using the FPGA (Field Programmable Gate Array) technique. We also integrate the monitoring functionality in the prototype.

The remainder of the paper is organized as following. Section 2 first gives an overview of related work on simulation tools and investigations of novel cache architectures. This is followed by a brief description of the developed cache model in Sect. 3 with a focus on its specific feature and simulation infrastructure. Section 4 presents the experimental results showing how applications behave with different cache configurations. The hardware design and implementation is described in Sect. 5. The paper concludes in Sect. 6 with a short summary and future directions.

## 2 Related Work

Simulation is a common used approach for evaluating hardware design before the real hardware is implemented. A well-known example is the FLASH simulator [10] that models the complete architecture of the FLASH multiprocessor. In addition to such simulators of special purpose, tools for general research studies are also available. Representative products include SimOS [11], SIMICS [12], and SimpleScalar [13]. We choose SIMICS as an example for a deeper insight.

SIMICS is a full-system simulator with a special feature of allowing the booting of an original operating system to the simulation environment. It models shared memory multiprocessor systems and the execution of multi-threading applications. It also contains a module that models the basic functionality of caches. This module can be extended for individual purpose of different researchers. Due to this characteristics, a set of simulation-based research work in the field of cache memories are conducted on top of SIMICS. For example, Curtis-Maury et al. [14] use this simulation platform to evaluate the performance of OpenMP applications on SMP and CMP architectures in order to identify architectural bottlenecks. The L2 miss was used as a performance metric for this evaluation. Liu et al. [6] apply SIMICS to verify their proposed mechanism for implementing a kind of split organization of the second level cache.

Besides the full-system simulators, several specific cache models have also been developed. Cachegrind [15] is a cache-miss profiler that performs cache simulation and records cache performance metrics, such as number of cache misses, memory references, and instructions executed for each line of the source code. It is actually a toolkit contained in the Valgrind runtime instrumentation framework [16], a suite of simulation-based debugging and profiling tools for programs running on Linux. MemSpy [17] is a performance monitoring tool designed for helping programmers

to discern memory bottlenecks. It uses cache simulation to gather detailed memory statistics and then shows frequency and reason of cache misses.

In the area of reconfigurable cache systems, different architectures have been proposed. However, the verification was all based on simulation. Benitez et al. [18] proposed a reconfigurable cache architecture that can be adapted to the running program. The adaptation scheme is based on two techniques: a learning process provides the best cache configuration for each program phase, and a recognition process detects program phase changes. In addition, a low-overhead reconfiguration mechanism was designed. Gordon-Ross et al. [19] did similar work, but applied heuristic tuning method to determine the best cache parameters. Abella et al. [20] designed a hardware technique to turn off the cache lines that are not expected to be reused. Alternatively, Ishihara et al. [21] applied specific algorithm to place the code in the cache in a way that not all cache lines in a set are used. Unused cache lines can be disconnected for saving the power.

Overall, cache architecture has been studied using simulation in the last years and chip-multiprocessor has also been addressed. We follow this traditional approach, but conduct a novel investigation, i.e. studying the adaptivity between OpenMP performance and the cache organization. We even go further: designed and implemented a hardware prototype.

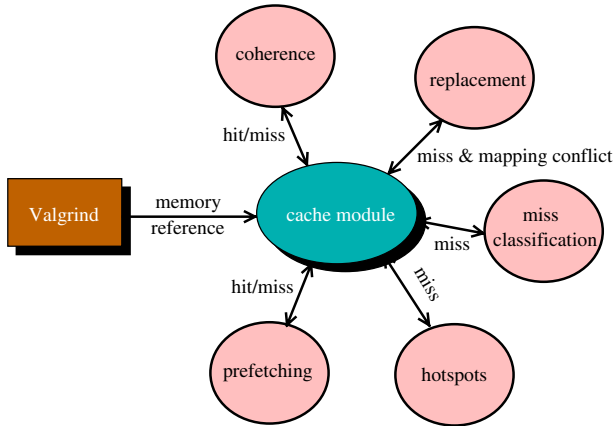
### 3 The Cache Simulator

As available simulators do not deliver all required properties, e.g. performance metrics, for this specific study, we developed a cache simulator called CASTOR (CAche SimulaTOR).

#### 3.1 CASTOR Features

First, CASTOR simulates the basic functionality of the cache hierarchy on a multicore machine. For flexibility, cache-associated parameters such as size, associativity, cache level, and write policies can be arbitrarily specified by users. Corresponding to most of the realistic cases, we assume that one processor runs only a single thread and the master thread, which is responsible for the sequential part of an application, is mapped to the first processor.

CASTOR simulates several cache coherence protocols for filling the different requirement of users. In addition to the conventional MESI scheme, other protocols like MSI and MEOSI are also supported. These protocols differ in the number of status of a cache line, where MESI scheme uses four status, Modified, Exclusive (unmodified), Shared (unmodified), and Invalid, to identify a cache line, while MSI applies only three of them and MEOSI deploys an additional one “Owner” to show the home node of the cached data. False sharing scenarios are identified in order to make this simulator more general that it can also be applied for other research work, for example, code optimization. For the same reason, mechanisms of classifying cache misses and various prefetching schemes are implemented within CASTOR.



**Fig. 1** CASTOR overall infrastructure

### 3.2 Simulation Infrastructure

CASTOR requires a frontend to deliver the runtime memory references of an OpenMP program. We could integrate CASTOR into the SIMICS environment, however, for this research work, a complicated full-system simulator is not necessary. In this case, we deployed Valgrind to provide the needed information and built an interface between CASTOR and this frontend for an on-line simulation. Alternatively, memory references can be stored in a trace file enabling simulations with different cache parameters, without having to run the application again.

CASTOR, on the top level, is composed of several modules. As shown in Fig. 1, the *cache module*, the main component of this infrastructure, is responsible for the basic functionality of a cache hierarchy. For each memory access from Valgrind it performs the search process in the caches available for the thread issuing this reference and thereby determines whether the access is a hit or a miss at each cache level. During this process, other modules could be initiated for additional functions. First, the *coherence module* is called to handle the cache coherence issues. In case of a mapping conflict, the *replacement module* is deployed to achieve a free cache line for incoming data. For a cache miss, the *miss classification module* is adopted to determine the miss type, which lies in four categories: compulsory, capacity, conflict, and invalidation miss. Compulsory misses occur when the data is first accessed, hence this kind of miss is also called first reference or cold miss. In this case, the data is still in the main memory and has to be loaded to the cache. Conflict misses occur when a data block has to be removed from the cache due to mapping overlaps. Capacity misses occur when the cache size is smaller than the working set size. The information about the miss type can give the programmers valuable help in code optimization. The *hotspot module* is responsible for mapping the cache miss to the functions in the source program. This feature allows both CASTOR to deliver simulation results at high-level and the users to find access bottlenecks. Finally, the *prefetching module* is called when the user has specified a kind of prefetching. In this case, appropriate data blocks are loaded to the cache and the cache state is correspondingly updated.

### 3.3 Performance Data

CASTOR provides at the end of simulation a set of performance metrics, such as statistics on cache hits, cache misses and each miss category, cache line invalidations, false sharing, and prefetching. Performance information can be given at the basis of the complete program or for each individual function contained in a program.

## 4 Experimental Results

Based on CASTOR, we could study the influence of different cache organizations on the performance of OpenMP applications. Six applications from both the SPEC [22] (Equake, Mgrid, and Gafort) and NAS [23,24] (CG, FT, and EP) OpenMP benchmark suite are tested. For smaller memory access traces the SPEC applications are simulated with the test data size and applications from the NAS Parallel Benchmark are configured with CLASS S.

### 4.1 Private or Shared L2?

The first study addresses the organization of the cache level closest to the main memory, i.e. the L2 cache. Applications are executed on systems with 2, 4, and 8 cores separately and the number of cache misses are measured. To allow a fair comparison, the size of the shared L2 cache is the sum of all local L2s in the private case. Cache parameters are chosen as: 32 bytes of cache line, 4-way write-through L1 of 16K, 8-way write-back L2 of 512K (private), MESI coherence protocol, and the LRU replacement policy that chooses the least recently used cache block as candidate to store the requested data. We select several interesting results for demonstration.

Figure 2 is the experimental result with the CG code. For a better understanding of the cache behavior, we show the miss in different classifications rather than only depicting the total misses which correspond to the height of the columns in the figure.

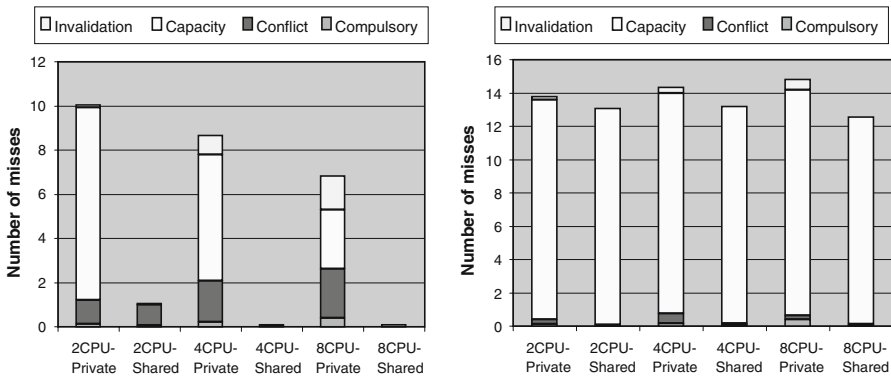


Fig. 2 L2 misses of the CG application

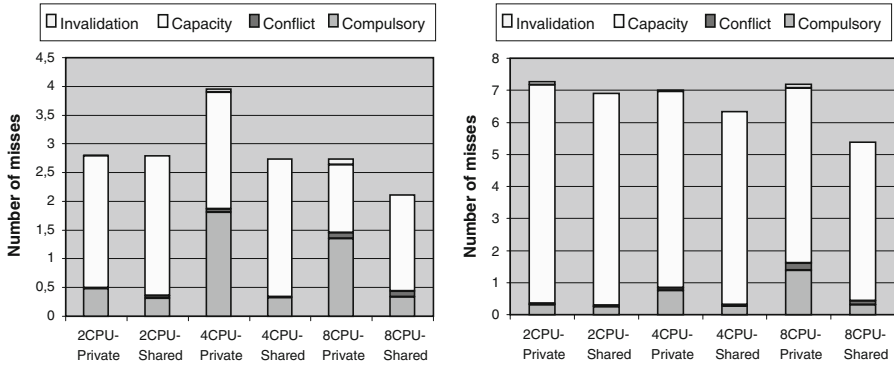


Fig. 3 L2 misses of the FT (left) and Equake (right) applications

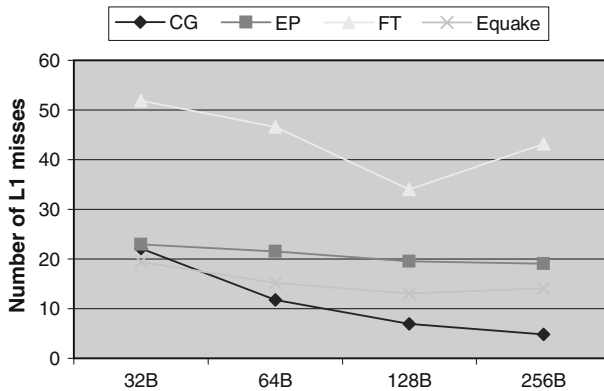
For this figure and all others in the following the number of misses is shown in a unit of one million.

As can be seen in the left diagram, a shared cache performs generally better than the private caches. In the case of 4 and 8 processors, almost all misses that occur with private cache are eliminated. This outstanding performance could be resulted by the relative large cache size, in contrast to the working set, and perhaps does not suffice for a general conclusion. Therefore, we simulated the code again with a smaller L2 of 64K. As shown in the right diagram of Fig. 2, the same result can be observed: shared L2 outperforms private caches.

Observing both diagrams, it can be seen that for the CG program capacity miss is the main reason of poor performance of private caches. It can also be seen, more clearly in the left diagram, that the performance gain achieved with shared L2 is also primarily due to the reduction of the capacity miss. This can be explained by the fact that in the shared case a larger L2 (in size of combined private L2s) is available to each processor and as a consequence capacity problem can be relaxed. However, the CG application also shows a reduction of conflict miss in the shared L2 cache. This feature depends on the access pattern of this program. As the data requested by a single processor has different mapping behavior in shared and private caches, conflicts that occur in a private cache could be removed with a larger shared cache.

Similar to CG, other applications also prefer to a shared L2. Figure 3 demonstrates the result with the FT and Equake program. For both codes, it can be seen that the better performance with a shared L2 is contributed by the reduction of compulsory misses. The capacity miss, another dominant miss category with these programs, however, can not be removed with a shared cache. This means the combination of the L2 caches on all processors can not achieve a cache memory that is large enough to hold the needed data for running these programs.

Overall, the experimental results depict that OpenMP applications benefit from the shared cache. Although different codes have their own contribution to this, several common reason exist. For example, parallel threads executing a program share partially the working set. This shared data can be reused by other processors after being loaded to the cache, reducing hence the compulsory miss. Shared data also saves space



**Fig. 4** L1 miss with changing block size

because it is only cached once. This helps to reduce the capacity miss. In addition, there are no invalidation misses with a shared cache.

#### 4.2 Static Reconfigurable Cache?

The second experiment aims at finding how the performance of an application varies across different cache configurations. For this we simulated the cache behavior of each application with various cache size, line size, and associativity. Number of total misses in the L1 cache were measured.

Figure 4 is the result with changing cache block size. It can be seen that applications differ in cache access behavior with this parameter. CG presents the best scalability, where cache miss keeps reducing as the block size increases. EP shows the same behavior, but with only slight reduction in cache miss. The other programs, FT and Equake achieve the best performance with a cache block size of 128B and further enlarging the blocks worsens cache performance.

To better understand these different behavior, we have examined the miss classification of two representative programs: CG and FT. We found that the good performance and scalability of CG is mainly contributed by the reduction of capacity miss, but also the conflict miss which maintains reducing even though only slightly. For the FT code, however, the conflict miss grows with 256B and the reduction in capacity miss can not compensate for this. As a consequence, more misses are caused with a 256B cache than a 128B one.

For changing associativity, nevertheless, the result is not so identical. As can be seen from Fig. 5, FT and EP have increasing cache misses with larger associativity; CG behaves similarly with various set size; Equake benefits significantly from the increasing associativity; and the rest two applications, Gafort and Mgrid, have a better performance with a 4-way cache but a further increment in associativity does not introduce more cache hits.

For explaining these different behavior, we examine further the miss classifications of all applications shown in Fig. 6.

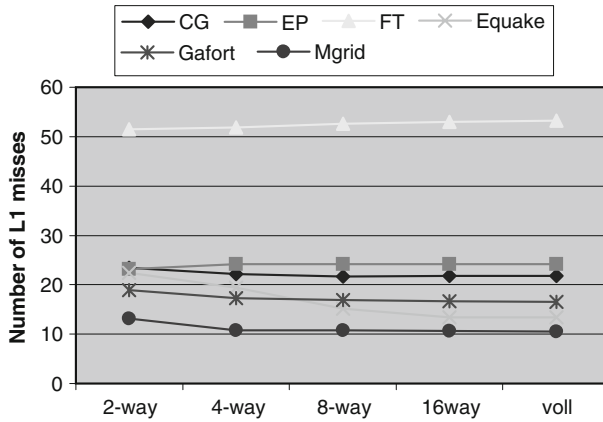


Fig. 5 L1 miss with changing associativity

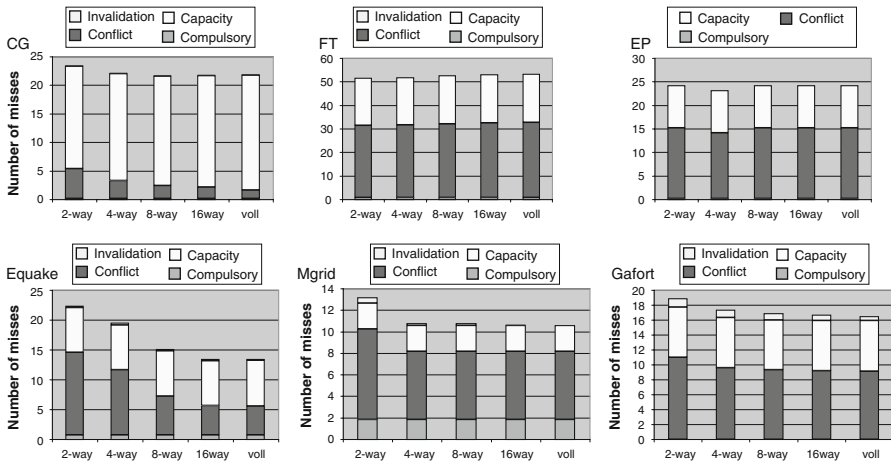
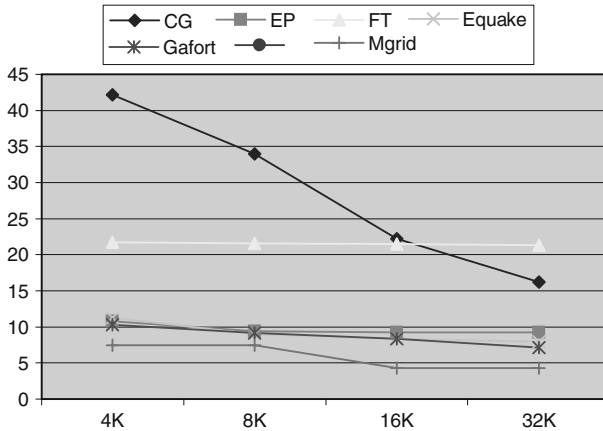


Fig. 6 Miss classification with changing associativity

For the CG program, it can be seen that the conflict miss, as expected, decreases with the increasing number of blocks within a cache set. However, the capacity miss, another dominating miss of this code, grows up with the associativity. Combined, the cache performance gets worse.

For the FT program, nevertheless, conflict misses increase with the associativity. This behavior is not expected. Hence, we have examined the three primary individual functions in this program and observed that one of them has less miss with larger cache sets but the others perform best with a 2-way cache. This means an individual tuning with respect to the functions, e.g. using a reconfigurable cache architecture capable of changing the configuration at the runtime, could be needed by this application.

EP shows less miss with a 4-way cache than a 2-way cache, but a further increase in associativity introduces more cache misses. It can be observed that EP has capacity problem. However, the capacity miss does not change across different set size.



**Fig. 7** L1 miss with changing cache size

This specific behavior lies singly on the conflict miss and can only be explained by the individual access pattern of this application.

Equake achieves the best performance with the changing associativity: number of cache misses keeps going down up to full-associative caches. As shown in Fig. 6, this is completely contributed by the reduction of conflict miss. Hence, increasing the blocks of a cache set helps this application to tackle the mapping conflict.

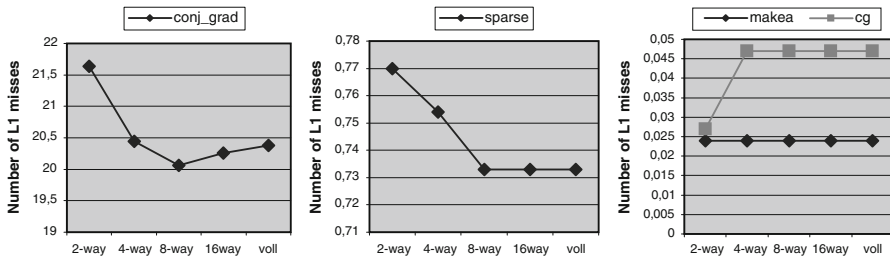
Mgrid and Gafort show a significant miss reduction when the associativity switched from 2-way to 4-way. However, further enlarging the cache set does not introduce much change in cache performance. This means for both applications a 4-way cache is preferred.

For changing cache size, as shown in Fig. 7, no worse behavior can be seen with the increase of the cache capacity. Applications differ in this behavior in that some benefit more and others less. For example, CG needs a large cache for a high cache ratio, but for FT a larger cache can not remove the misses significantly.

In summary, OpenMP applications have different requirement in cache organization. For example, a specific line size can significantly improve the cache performance of an individual application, but introduces no gain with another program. A large cache can potentially considerably reduce the cache miss of one application, but is not necessary for another. This means a static tuning of the cache configuration according to the requirement of individual applications can result in better performance and at the same time efficiently utilize the cache resource.

### 4.3 Dynamic Reconfiguration?

The last experiment aims at examining the cache behavior of different functions within a single program. For this, we use the *hotspots* component of CASTOR to order the overall cache misses to individual functions. Again we measured the number of total L1 misses.



**Fig. 8** L1 miss of functions with changing associativity

Figure 8 is a sample result achieved with the CG code. Three diagrams in the figure depict the changing behavior, with respect to the cache associativity, of the four functions with the most cache misses. It can be seen that the best set size for both *conj\_grad()* and *sparse()* is 8, while for *makea()* a 2-way cache is better and *cg()* has no specific need. This indicates that a dynamic tuning of the cache line size potentially improves the performance of CG. As mentioned in the previous subsection, FT also shows this feature. However, for other applications and cache parameters, no considerable distinction has been observed. Hence, for these applications a statically reconfigurable cache suffices.

## 5 Hardware Prototype

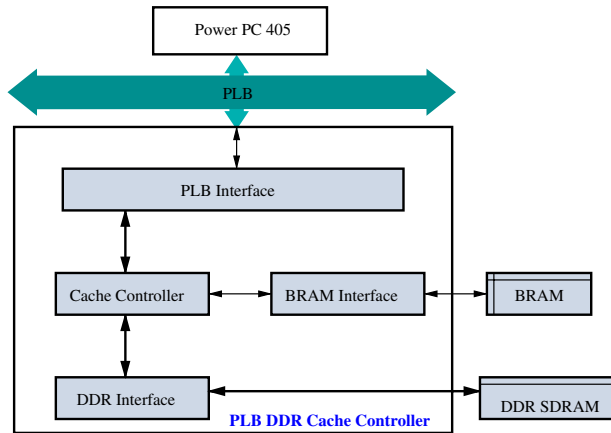
The reconfigurable cache [25, 26] is implemented using a Xilinx Virtex 4 FPGA (Field Programmable Gate Array), which is configured with a Power PC 405 processor core and a Power Local Bus (PLB). The main memory, a DDR-SDRAM, lies on an additional chip. Our task and goal is to implement a cache to buffer the data from the SDRAM.

The first design issue is to decide where to place the cache module on the FPGA. One choice is the PLB which holds a connection to the SDRAM. This position has the advantage that the cache access latency are not influenced by other modules. However, the PLB controller must be modified. Hence, we determined to implement an individual cache controller which is independent of both the PLB and the SDRAM.

As depicted in Fig. 9, the cache controller lies between the PLB and the DDR-SDRAM. Through the PLB interface, signals from and to the bus can be captured by the cache controller. For a connection to the DDR-SDRAM, a DDR interface is available. Buffered data are stored in the BRAM (block RAM), which is the storage part of the reconfigurable cache system.

In addition, we integrate a monitor module on the chip to trace the cache events, and the instruction and data address of these events. A small ring buffer is available on the monitor module for storing the monitoring information, but the data are moved to the SDRAM when the buffer is full. The monitoring information is used to find cache problems and further to propose a better cache organization.

Core of this work is the cache controller. This module is responsible for controlling the basic cache functionality and the reconfigurability. For the former we designed a



**Fig. 9** Chip structure with the cache controller

state machine to process the memory references which can be a read or a write. For read operations it is examined if valid data can be found in the BRAM and in case of a replacement some data are potentially written back to the SDRAM. Similarly, for write operations the DRAM is sought for valid data and a write to the SDRAM can be followed depending on the applied write policies.

For reconfiguration we designed a communication and synchronization protocol which controls the initialization, write-back, and reordering of the cache. Several registers are deployed to store the configuration parameters coming from the processor. Based on the given value of the parameters, the length of *tag* and *set* is calculated and the cache is reordered. The processor is informed with signals when the reconfiguration is done.

Cache reordering is a specific feature of our design. This mechanism aims at possibly holding the data in the cache for reuse when the configuration changes. Actually, we can generally invalidate the whole cache and write the dirty lines back to the memory. However, this causes a significant performance loss because the empty cache has to be warmed up. Our protocol therefore handles specific individual configuration cases where useful data can be maintained with cache reordering. For example, when doubling the line size but halving the associativity, the whole cache data can be saved for reuse because the new configuration is achieved by combining two neighbouring cache lines and in this case the set number does not change.

## 6 Conclusions

Multicore multiprocessor is the trend of microprocessor design and OpenMP is an appropriate programming model for it. As the memory wall keeps widening, cache performance becomes increasingly important, especially for multicore processors with potentially more cache misses.

This paper investigates reconfigurable cache architectures and their impact on the performance of OpenMP programs. The study is based on a self-written cache

simulator that comprehensively models the cache functionality and provides thereby performance metrics for evaluating the cache access behavior. We have focused on the structure of the second level cache and issues with reconfigurable caches. The achieved results have directed us to design and develop a hardware prototype of this novel architecture.

However, this is only an initial step towards our final goal of runtime cache tuning according to the access pattern of running applications. In the next step, we have to design algorithms for analysing the monitoring data to detect the access feature and further to compute an optimal cache configuration. In addition, instruction sets must be extended with additional instructions for issuing the reconfiguration signal.

## References

1. Chandra, R. et al.: *Parallel Programming in OpenMP*. Number 978-1-55860-671-5 in ISBN. Morgan Kaufmann (2000)
2. Pacheco, P.: *Parallel Programming with MPI*. Number 978-1-55860-339-4 in ISBN. Morgan Kaufmann (1996)
3. Fung, S.: *Improving Cache Locality for Thread-Level Speculation*. Master's thesis, University of Toronto (2005)
4. Wang, Z., Sha, E., Hu, X.: Combined partitioning and data padding for scheduling multiple loop nests. In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 67–75 (2001)
5. Somnath, G., Margaret, M., Sharad, M.: Precise miss analysis for program transformations with caches of arbitrary associativity. *ACM SIG-PLAN Notices* **33**(11), 228–239 (1998)
6. Liu, C., Sivasubramaniam, A., Kandemir, M.: Organizing the last line of defense before hitting the memory wall for CMPs. In: *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'04)*, pp. 176–185, Madrid, Spain, February 2004
7. Molnos, A.M., Cotofana, S.D., Heijligers, M.J.M., van Eijndhoven, J.T.J.: Static cache partitioning robustness analysis for embedded on-chip multi-processors. In: *Proceedings of the 3rd Conference on Computing Frontiers (CF'06)*, pp. 353–360, Ischia, Italy, May 2006
8. Benitez, D., Moure, J.C., Rexachs, D.I., Luque, E.: Evaluation of the field-programmable cache: performance and energy consumption. In: *Proceedings of the 3rd Conference on Computing frontiers (CF'06)*, pp. 361–372, Ischia, Italy, May 2006
9. Carvalho, M.B., Goes, L., Martins, C.: Dynamically reconfigurable cache architecture using adaptive block allocation policy. In: *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006
10. Gibson, J., Kunz, R., Ofelt, D., Horowitz, M., Hennessy, J., Heinrich, M.: FLASH vs. (simulated) FLASH: closing the simulation loop. In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 49–58, November 2000
11. Herrod, S.A.: *Using Complete Machine Simulation to Understand Computer System Behavior*. Ph.D. thesis, Stanford University, February 1998
12. Magnusson, P.S., Werner, B.: Efficient Memory Simulation in SimICS. In: *Proceedings of the 8th Annual Simulation Symposium*. Phoenix, Arizona, USA, April 1995
13. Austin, T., Larson, E., Ernst, D.: SimpleScalar: an infrastructure for computer system modeling. *Computer* **35**(2), 59–67 (2002)
14. Curtis-Maury, M., Ding, X., Antonopoulos, C., Nikolopoulos, D.: An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In: *Proceedings of the First International Workshop on OpenMP (IWOMP)*, Eugene, Oregon USA, June 2005
15. WWW.Cachegrind: a Cache-miss Profiler. Available at [http://developer.kde.org/sewardj/docs-2.2.0/cg\\_main.html#cg-top](http://developer.kde.org/sewardj/docs-2.2.0/cg_main.html#cg-top)
16. Nethercote, N., Seward, J.: Valgrind: a program supervision framework. In: *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003. Available at <http://developer.kde.org/sewardj>

17. Martonosi, M., Gupta, A., Anderson, T.: Tuning memory performance of sequential and parallel programs. *Computer* **28**(4), 32–40 (1995)
18. Benitez, D., Moure, J.C., Rexachs, D.I., Luque, E.: Evaluation of the field-programmable cache: performance and energy consumption. In: *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pp. 361–372 (2006)
19. Gordon-Ross, A., Vahid, F., Dutt, N.: Fast configurable-cache tuning with a unified second-level cache. In: *ISLPED '05: Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pp. 323–326 (2005)
20. Abella, J., González, A., Vera, X., O'Boyle, M.: IATAC: a smart predictor to turn-off L2 cache lines. *ACM Trans Arch Code Optim* **2**(1), 55–77 (2005)
21. Ishihara, T., Fallah, F.: A non-uniform cache architecture for low power system design. In: *ISLPED '05: Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pp. 363–368 (2005)
22. Saito, H. et al.: Large system performance of SPEC OMP2001 benchmarks. In Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M. (eds.) *High performance computing: 4th International Symposium, ISHPC 2002. Proceedings, Volume 2327 of Lecture Notes in Computer Science*, pp. 370–379, May 2002
23. Bailey, D. et al.: *The NAS Parallel Benchmarks*. Technical Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994
24. Jin, H., Frumkin, M., Yan, J.: *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Technical Report NAS-99-011, NASA Ames Research Center, October 1999
25. Nowak, F., Buchty, R., Karl, W.: Adaptive cache infrastructure: supporting dynamic program changes following dynamic program behavior. In: *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA 2008)*, Dresden, Germany, February 2008
26. Buchty, R., Nowak, F., Karl, W.: A Run-time Reconfigurable Cache Architecture. In: *Proceedings of the International Conference ParCo 2007, Volume 15 of Advances in Parallel Computing*, ISBN 978-3-9810843-4-4, pp. 757–766. IOS Press, Juelich, Germany, September 2007