Evaluating the Cache Architecture of Multicore Processors

Jie Tao[1], Marcel Kunze[1], and Wolfgang Karl[2]

[1]Steinbuch Centre for Computing [2]Institut für Technische Informatik
Karlsruhe Institute of Technology, Germany
jie.tao@iwr.fzk.de

Abstract

Microprocessor architecture for both commercial and academical purpose is coming into a new generation: multiprocessors on a chip. Together with this novel architecture, questions and research topics also arise. For example, how to design the on-chip caches to avoid memory operations becoming the performance bottleneck?

In this work, we study the impact of various cache architectures on the execution behavior of multi-threading applications. We focus on four general design issues: cache structure, configuration parameters, coherence influence, and prefetching strategies. The study is based on a self-developed cache simulator that models the functionality of a multicore cache hierarchy with arbitrary levels and various organizations. The achieved results can direct both hardware and program developers to optimize their cache designs or the program codes.

Keywords: Cache performance, Multicore processor, Simulation, OpenMp application

1 Introduction

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.

However, this high performance is based on an assumption that the main memory does not stall the processors for acquiring the requested data. In a real word, nevertheless, this can not be guaranteed. First, on multicore architectures the per-processor cache is smaller in contrast to those provided by uniprocessor or SMP machines. 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 be more expensive. Hence, how to adapt the cache architecture to applications, or vice versa, forms a new challenging research area for both architecture designers and application developers.

This work aims at examining such adaptivity. More concretely, we study various cache designs and their influence on the program execution. We focus on cache structures, configuration parameters, coherence effects, and prefetching strategies. The first study aims at determining whether a specific cache architecture, e.g. a private cache or a shared organization, introduces better performance than the other. The second investigation targets on cache configuration with the goal of examining how the performance varies with changing cache size, block size, and associativity. This study also aims at giving researchers in the area of reconfigurable cache architecture quantitative information about the potential and benefit of such novel designs. In addition, we study cache coherence effects to observe whether coherence is a possible performance bottleneck. We also address prefetching, a pre-loading technique often integrated in the cache hardware, and compare the cache performance with different prefetching schemes.

For enabling theses studies, a simple, flexible cache simulator was developed. This simulator models the basic functionality of a multicore cache hierarchy, and additionally contains a set of mechanisms for classifying cache misses, detecting access hotspots, and tracing the cache events.

The remainder of the paper is organized as follows. Section 2 describes the cache simulator in detail. This is followed by initial experimental results in Section 3. Section 4 gives a brief description of several related work in the area of cache architecture designs of multicore processor systems. The paper concludes in Section 5 with a short summary of this research work.

0-7695-3089-3/08 $25.00 © 2008 IEEE
DOI 10.1109/PDP.2008.22

IEEE computer society

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

2 Multicore Cache Simulation

Simulation is a conventional approach to understand the execution behavior of applications and thereby to evaluate the hardware design of a computer system. As a consequence, a number of simulators have been developed, either for special purpose or for general research studies. For the former, a well-known example is the FLASH simulator [5] which models the complete architecture of the FLASH machine. For the latter, SimpleScalar [1] and SIMICS [8] are the most frequently applied tools, where SimpleScalar is usually used for evaluating processor techniques and SIMICS for the memory and operating system. Similarly, we developed a cache simulator for this study,

The Front-end. First, a cache simulator needs a frontend to acquire the runtime memory references of an application. We could apply the SIMICS simulation environment and integrate the cache simulator into its cache module, however, for this research work, a complicated full-system simulation tool is not necessary. For simplicity and less slowdown we deployed the Valgrind runtime instrumentation framework [11] to provide the needed information.

Valgrind is a suite of simulation-based debugging and profiling tools designed for performance study of programs running on Linux systems. Actually, Valgrind contains a cache-miss profiler, called Cachegrind, that performs cache simulation and records cache performance metrics. However, Cachegrind does not model multiprocessor caches and only provides statistics on several global events, like cache miss and memory reference. For achieving more performance metrics that allow us to conduct a comprehensive study of the multicore cache hierarchy, we developed this cache simulator and built an interface to Valgrind for gathering memory references.

Simulation Infrastructure. The basic functionality of this simulator is to model the organization and data access of a cache hierarchy on multicore processor systems. For flexibility, cache-associated parameters such as size, associativity, cache level, and write policies can be arbitrarily specified by the user. 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.

Besides the basic functionality, the cache simulator has additional mechanisms for analyzing the feature of cache misses, tracing the cache event, performing data prefetching, and detecting access hotspots.

Overall, our simulator is composed of several modules. Figure 1 illustrates the global infrastructure. The main component of this infrastructure is the Cache-Module which is shown in the middle of Figure 1. For each memory access from Valgrind it performs the search process in the cache

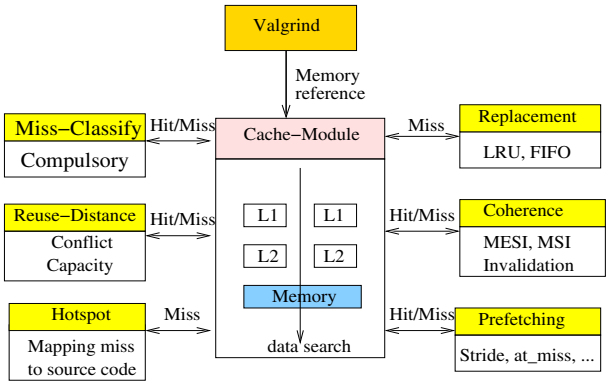

Figure 1. The overall simulation infrastructure

hierarchy and determines thereby whether this access is a hit or a miss. During this process, other modules are called for additional activities. First, if the access is a miss at a cache level and the mapping set in this cache is full , the Replacement module is called to select a cache block for eviction. In case of a write access, the Coherence module is initiated for executing the selected coherence protocol. The Prefetching module is only deployed when a prefetching strategy is specified by the user. The rest components, which are depicted on the left side of Figure 1, are used to classify the cache misses and detect access hotspots.

Cache Coherence. Currently, we simulate two coherence protocols: MESI and MSI. As mentioned, the Coherence module is responsible for conducting these protocols. By a memory write, this component checks possible copies of the associated data block in all processors and performs invalidation on the findings. Based on the access address, it also detects false-sharing scenarios.

Miss Classification. Cache miss on a multiprocessor machine can be distinguished between four types: compulsory, conflict, capacity, and invalidation. Compulsory misses are caused by referencing a data block at the first time; conflict misses occur when several blocks compete for the same caching place; capacity misses exist when the cache is not sufficient for holding the complete working set; and the invalidation miss is resulted by coherence activities. To better understand the cache access behavior it is needed to know the miss type. Therefore, we integrate this functionality into the cache simulator.

The Miss-Classify module in Figure 1 is actually only responsible for recognizing the compulsory miss. We use a hash table to store the first references and for each cache miss this table is scanned to determine if it handles with a compulsory miss.

The Reuse-Distance module is responsible for detect-

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

ing the rest type of cache misses. The main task of this component is to compute reuse distance and set reuse distance, which are factors for classifying conflict and capacity miss. Reuse distance of a memory reference is the number of unique data blocks between this reference and the last access to the same data block. Set reuse distance is similarly defined but counts only data blocks that lie in the same cache set. If the reuse distance is smaller than the number of cache blocks and the set reuse distance is larger than cache associativity, a miss is regarded as a conflict miss. Otherwise, the miss is a capacity miss. However, before ordering a cache miss to the conflict or capacity category, invalidation miss must be first checked. If a cache miss is caused by the fact that the requested data was invalidated and would not be evicted from the cache due to either conflict or capacity problems, the miss is counted to an invalidation miss. Here, a miss that is produced via conventional replacement is not considered as invalidation miss, although an invalidation has been performed on the data.

Prefetching. We simulate five different prefetching schemes. *Always* prefetches the successive data block of the currently requested data by each access. This scheme is often applied by hardware prefetching mechanisms. *On-miss* prefetches the next data block only when the current access causes a cache miss. With the *Tagged* [12] strategy, only references to data blocks, which are marked with a prefetching bit, cause prefetching of their succeeding. *Stride* [10] aims at prefetching the data that is really needed. This scheme applies a global history buffer to follow the cache misses and observes thereby whether the accesses issued by the same instruction have a constant stride. If such a constant exists, the prefetching follows with this stride. However, there exist access patterns that do not hold a constant stride but can be predicted. *Delta* [10] improves the *stride* scheme at this point. It correlates the successive access pairs and prefetches data according to the pattern of the so-called delta-pairs.

Access Hotspots. Based on both the instruction address associated with a cache miss and the debugging information, cache misses are ordered to individual functions in the source code. This not only enables a deeper insight into the cache access behavior, but also the finding of access bottlenecks.

3 Initial Experimental Results

The developed simulator gives us sufficient information to examine the cache behavior of an application with respect to both the whole program and single subroutines. We simulated six applications from the SPEC [4] and NAS [3, 6] OpenMP benchmark suite. For avoiding large 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.

L2 Structure. The first study addresses the organization of the cache level closest to the main memory. According to the current standard design, L2 is the last on-chip cache. We executed the applications on systems with 2, 4, and 8 cores separately and with both private and shared L2 organization where in the private case each processor core has a local second level cache and in the shared case a combined L2 is available for all cores. To allow a fair comparison, the size of the shared L2 is the sum of all local L2s in the private case. Cache parameters are chosen as: 32 bytes of line size, 4-way write-through L1 of 16K, 8-way write-back L2 of 512K (private), MESI coherence protocol, and the LRU replacement policy.

Figure 2 is the experimental result with all programs. Within this figure, the cache miss is shown using a colored bar which shall contain four segments. Each segment shows the number of misses (in a unit of one million) for a miss category. From top to bottom they represent the invalidation miss, capacity miss, conflict miss, and compulsory miss. As the miss number for a specific category could be zero or very small, most bars only have two or three segments.

Observing all diagrams, it can be seen that a shared cache performs generally better than private caches. This is especially clear with CG, Gafort, Mgrid, and FT on larger systems. For the CG code, for example, almost all misses that occur with the private cache are eliminated with the shared case. Observing the concrete diagram with CG, it can be seen that capacity miss is the main reason of its poor performance with private caches. It can also be seen that the performance gain achieved with shared L2 is primarily due to the reduction of the capacity miss. This can be explained by the fact that in the shared case a larger L2 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, EP performs better with shared cache also owing to the decrease in capacity miss. For FT and Equake, however, the better performance with a shared L2 is contributed by the reduction of the compulsory miss. It can also be seen that less misses are generated on larger systems, for example, a system with 8 cores. As mentioned, capacity miss is caused by first referencing a data block. It is clear that for private caches more processors in a system result in also more first reference misses. Hence, the compulsory miss arises with the number of processors. For a shared cache, however, processors share data and the shared data, which has been accessed by one processor, does not introduce compulsory miss when requested again by another

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

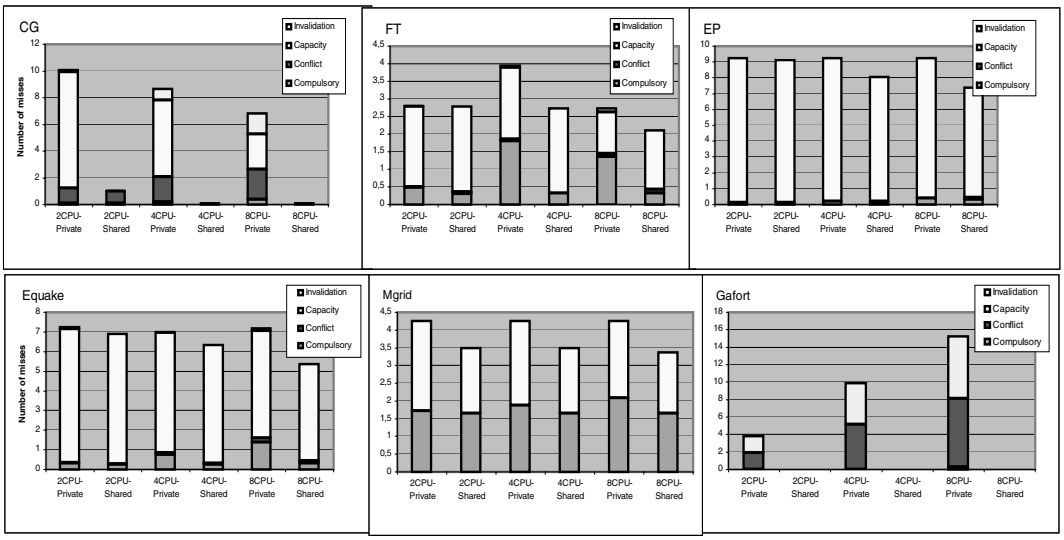

Figure 2. L2 misses of all tested applications

processor. If a program has a high share degree, like the FT program, compulsory miss can be significantly decreased in this way. Similarly, Mgrid also shows less compulsory miss with shared L2 caches. In addition, less capacity miss is another contribution to the better performance of a shared cache with this application.

Gafort depicts the best behavior with shared L2 among all tested programs. This is contributed by the elimination of both capacity and conflict miss.

Overall, the experimental results depict that applications benefit from the shared organization. Although different codes have their own contribution to this, several common reasons 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 because it is only cached once. This helps to reduce the capacity miss. In addition, there are no invalidation misses with a shared cache.

Configuration Parameters. Now we examine how different cache configurations influence the performance. For this, we simulated all applications again, but with various associativity, cache size, and block size. We measured the cache miss with the first level cache.

Figure 3 is the result with changing associativity. It can be observed that applications vary in cache behavior with this parameter. FT gets worse as the associativity grows. EP and CG perform first better with larger cache sets but then worse. The other programs, Equake, Gafort, and Mgrid, show less misses with increasing associativity. For explaining these different behavior, we take a deeper insight into the miss classification of all applications shown in Figure 4.

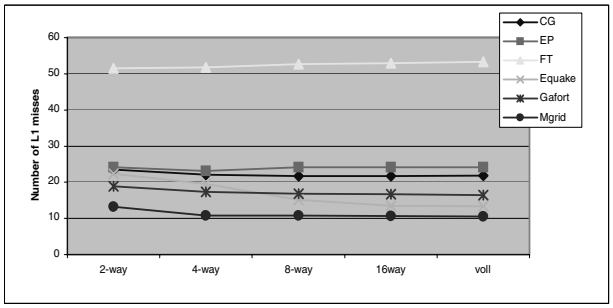

Figure 3. L1 miss with changing associativity

The CG program shows two dominating cache miss: capacity and conflict. The conflict miss, as expected, decreases with the increasing number of blocks within a cache set. However, the capacity miss grows up with the associativity. Combined, the cache performance gets worse.

The FT program also mainly suffer from capacity and conflict miss. Unlike CG, the conflict miss increase with the associativity for this application. This behavior is not expected. For exploiting the reason we examine further individual functions in this program. Figure 5 depicts the three cache critical functions. It can be observed that *evolve* has less miss with larger cache sets. Nevertheless, *fftz2* and *fffts3* 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.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

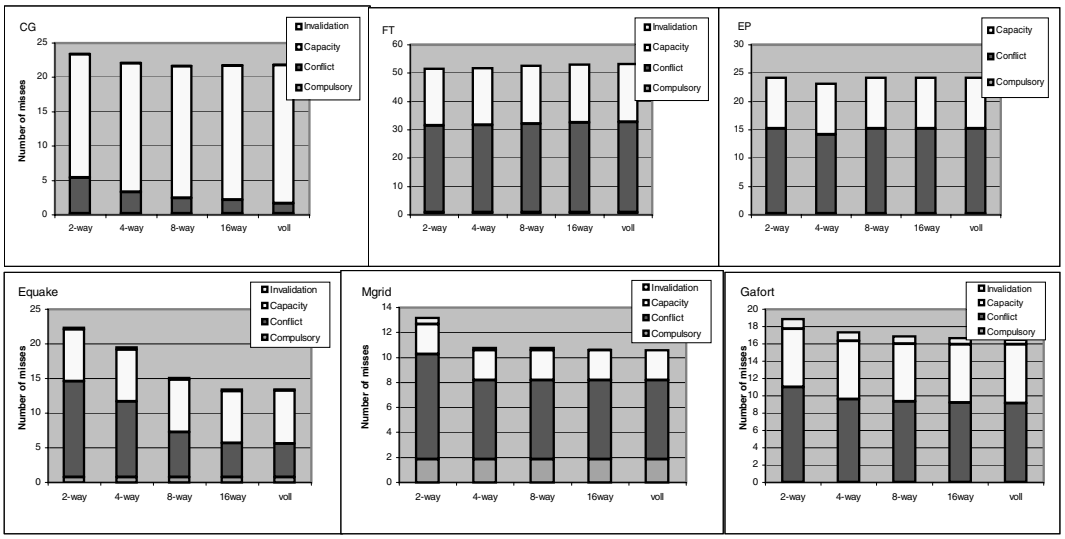

Figure 4. Miss classification with changing associativity

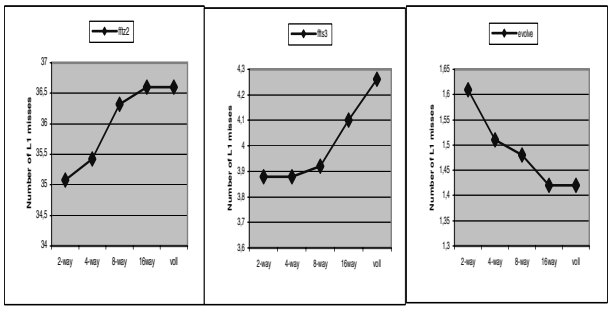

Figure 5. FT: insight into the functions

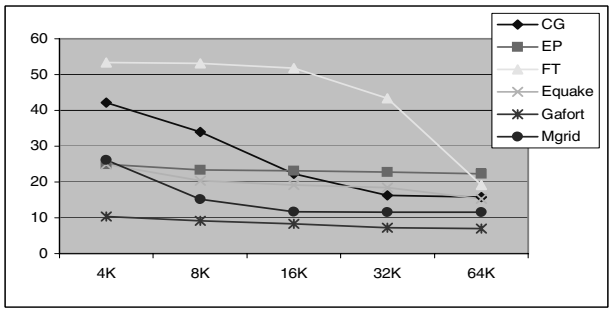

Figure 6. Miss with changing cache size

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

Equake achieves the best performance with the changing associativity: number of cache misses keeps going down up to full-associative caches. As shown in Figure 3, 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.

In summary, every application has its own requirement on cache associativity. This means, if the cache configuration can be tuned according to this individual need, all applications will benefit. Some applications, like FT, needs even a dynamic adaptation at the runtime.

Now, we observe the cache behavior with changing capacity. As shown in Figure 6, for all applications a larger cache is better than a smaller one. However, some applications benefit more, while some less. For example, the FT program has an extreme good performance with a 64K cache. Hence, for the chosen working set, FT requires a 64K L1 cache. However, having a deeper insight into individual functions, as shown in Figure 7, it can be observed that *fftz2* is the main contributor of the good performance. For other functions a large cache, which also means high energy consumption and management overhead, is not suggested. Again, FT needs specific tuning with individual functions in terms of cache size.

Figure 6 also depicts the different requirement of appli-

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

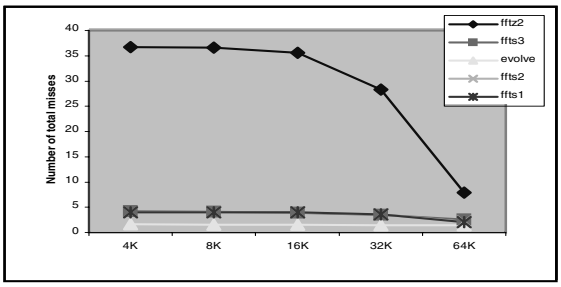

Figure 7. FT insight into functions

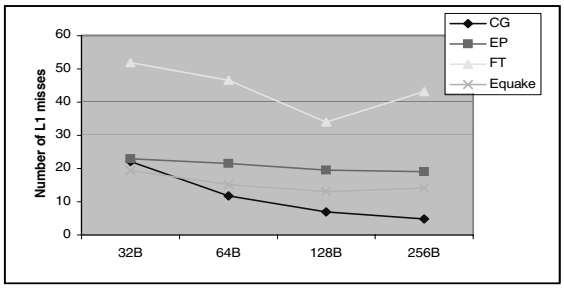

Figure 8. Miss with changing block size

cations. For CG a 32K cache is better, while Mgrid needs only a 16K cache. This indicates that, similar to associativity, cache size shall also be adapted to single applications.

In addition, it is interesting to see that the increase in cache size not only reduces the capacity miss, but also the conflict miss. For Mgrid, for instance, the conflict miss with a 8K cache is only 41% of that with 4K caches.

Another cache configuration parameter is the block size. As shown in Figure 8, applications behave differently 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, Equake, and Mgrid, all 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 examine the miss classification of three representative programs: CG, FT, and Mgrid. As shown on the left side of Figure **??**, the good performance and scalability of CG is mainly contributed by the reduction of capacity miss. In addition, the conflict miss with this code also reduces with the block size 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 cache misses are caused with a 256B cache than a 128B one. With Mgrid,

	Mgrid	Equake	Gafort	FT	CG	EP
Write-rate	6.06	17.5	18.6	17.7	14.5	29.8
Invalidation-rate	0.004	0.019	1.48	0.005	0.37	0.0006
True-sharing	92.7	87	95.4	97.5	92.2	97
Miss-percentage	69.1	81.2	35.8	21.3	38.6	18.2

Table 1. Rate of writes, invalidation, true sharing, and miss (in %)

the compulsory miss is reduced together with the capacity miss. Hence, a 64B cache behaves better than a 32B cache. Nevertheless, conflict miss grows up with larger blocks, resulting in a worse performance with 128B and 256B caches.

Overall, cache configuration parameters, especially the associativity and block size, have different impact on the cache access behavior of applications. Based on the simulation results, we can conclude that applications benefit from a reconfigurable cache architecture.

Cache Coherence. On multiprocessor systems, multiple copies of the same data block in the local cache of the processors must be maintained coherent. Typical coherence protocols use the invalidation strategy to achieve this. However, invalidation results in coherence miss which could be a performance bottleneck. For studying the impact of cache coherence on multicore processors, we measured a few coherence associated metrics. Table 1 is the result.

The first metric is the Write-rate which is the proportion of write operations to the total memory accesses. Since only writes could cause invalidations, we use this metric to observe the potential coherence traffic.

Invalidation-rate in the table is the percentage of total invalidations to the number of writes. This metric is used to predict the shared degree of an application. The values of True-sharing are computed with the number of true sharing invalidations divided by the total invalidations. It has to be noted that Miss-percentage in the table is not the invalidation miss rate, which has been applied in the above miss classification diagrams, rather it is the percentage of invalidation misses to the total invalidations.

Observing the Write-rate in Table 1, it can be seen that all applications have less writes than reads, where for most applications this value is below 20%. This is a positive factor for lower coherence traffic. Much better, theses write operations do not result in excessive invalidations. For example, the Gafort depicts the highest Invalidation-rate of only 1,48%. This indicates that coherence can not form bottlenecks for the tested applications. Actually, the low invalidation miss rate shown in the previous diagrams has presented this feature.

The high True-sharing rate indicates that invalidations are required, because processors use the same data word of a

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

shared copy. However, observing the Miss-percentage it can be seen that for some applications most invalidations do not cause cache miss. For example, only 18.2% of the invalidations with EP results in misses. This means the invalidated data block is either not further accessed or conventionally replaced by other blocks. Therefore, for applications with lower Miss-percentage it would be better to perform the invalidation when necessary, rather than at the time shared data is updated.

Prefetching. The last experiment aims at comparing different prefetching schemes and detecting whether there exists a best one for multicore processors. For this experiment, we chose the shared L2 as the target in order to exclude the influence of cache line invalidation.

Figure 9 depicts the experimental results. As Gafort has no conflict and capacity miss with a shared second level cache, the result with this code is not interesting.

It can be observed that for all applications prefetching reduces cache misses. The best improvement is achieved by *always* and *tagged*, where in some cases the former is better and in some others the latter is better. In a real system, *tagged* should be more efficient because according to our simulation results, *always* performs much more prefetches than *tagged* indicating a higher prefetch overhead.

Surprisingly, *stride* and *delta*, which prefetch according to the access pattern, are the worst algorithms for the tested applications. It could be caused by the fact that these applications do not hold many regular patterns and hence not sufficient prefetching is conducted.

Actually, prefetching is an appropriate technique for tackling compulsory miss. This has been proven by the experimental results. However, Figure 9 also depicts a reduction of other misses. With CG, for example, conflict is the dominating reason of cache misses and prefetching reduces this miss. FT, EP, and Equake show significant capacity miss and as a consequence, less capacity miss occur with prefetching. Mgrid shows a large number of compulsory and capacity miss, therefore, *always* and *tagged* reduces both misses.

Overall, prefetching generally decreases the number of cache misses. However, it is not clear whether this also results in a better execution time of applications. The reduction of cache miss is combined with the overhead of additional instructions for prefetching. We can conclude that *tagged* is the best algorithm for achieving less cache miss, but we can not claim that it introduces the best cache performance. The latter is a trade-off between the prefetching overhead and the gain, and has to be measured on real systems.

4 Related Work

With the trend of microprocessor towards multicore, increasing research work addresses on the cache design of this novel architecture.

Liu et al. [7] proposed a new architecture called Shared Pocessor-Based Split L2. With this design, the shared L2 cache is comprised of multiple smaller units which can be selectively allocated to the processor cores. This architecture has the advantage of configuring L2 caches based on application workload characteristics. The architecture is evaluated on the SIMICS simulation environment. The experimental results show a 11.52% average improvement in IPC over the private organization.

Modarressi et al. [9] designed a reconfigurable cache architecture for object-oriented application-specific instruction set processors (ASIP). With this design, a cache architecture is virtually divided into a number of partitions and the partition sizes can be dynamically changed depending on the run-time behavior of the application. The goal of cache partitioning is to both provide the concurrent memory access for multiple functional units and to reduce the number of tag comparisons per cache access. In addition, a simple and energy-efficient cache consistency mechanism was also developed. The impact of the proposed cache architecture on the cache energy consumption has been evaluated. Experimental results show that the proposed cache architecture reduces the number of tag comparisons per cache access by 39% on average.

Benitez et al. [2] designed a Field-Programmable Cache Array that can be dynamically reconfigures. A runtime algorithm was also developed to manage this specific architecture. This algorithm can compute the best cache configuration for each program phase, where program phase is detected using data working-set signatures.

In summary, researchers are investigating various cache designs for enhancing the performance of this component towards a high performance of the overall system. Our goal is to evaluate different traditional designs available on current microprocessors and thereby find potential combination or improvement for the emerging multicore systems. The self-developed cache simulator allows us to comprehensively understand the cache access behavior of applications with various cache designs.

5 Conclusions

This paper introduces our research work of applying a simulator to evaluate the cache architecture of multicore processor systems, with a focus on the L2 structure, configuration parameters, cache coherence, and prefetching. We found that a shared second level cache generally performs better than a private one located on each processor core.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.

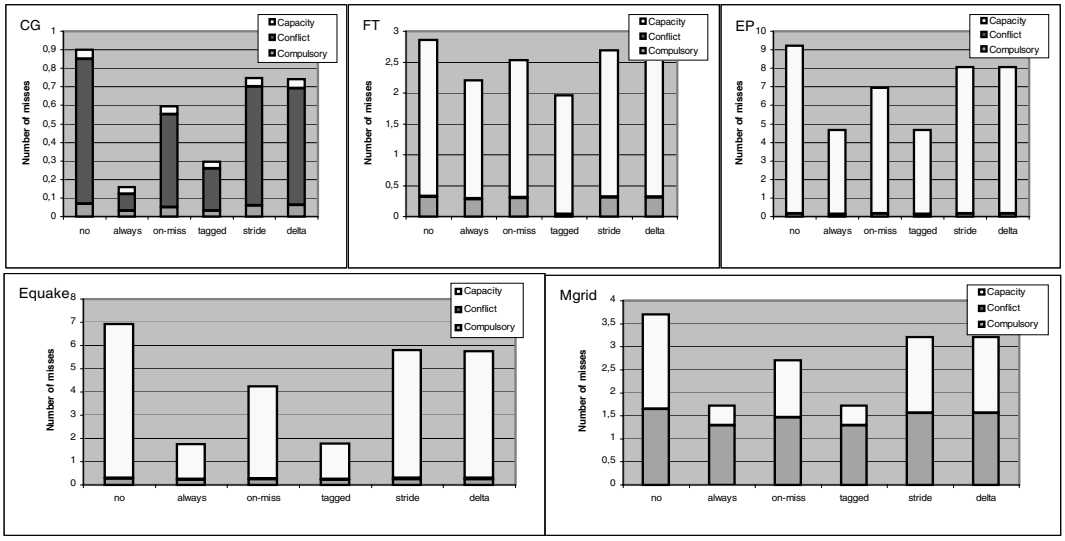

Figure 9. Cache miss with different prefetching schemes

For cache configuration, applications depict different behavior with changing associativity, cache size, and block size. Even individual functions within a single program behave differently with various configuration parameters. We conclude that a reconfigurable cache architecture capable of adapting to the individual requirement of different applications is potentially a preferred design for general-purposed multicore processors. For cache coherence, none of the tested programs show performance bottleneck with it. For the last study we found that prefetching generally reduces cache misses and some schemes outperform others for all applications in terms of the cache miss number.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[2] D. Benitez, J. C. Moure, D. I. Rexachs, and E. Luque. Evaluation of the Field-programmable Cache: Performance and Energy Consumption. In *Proceedings of the 3rd conference on Computing frontiers (CF'06)*, pages 361–372, Ischia, Italy, May 2006.

[3] D. B. et. al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994.

[4] H. S. et al. Large System Performance of SPEC OMP2001 Benchmarks. In H. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, editors, *High Performance Computing : 4th International Symposium, ISHPC 2002. Proceedings*, volume 2327 of Lecture Notes in Computer Science, pages 370–379, May 2002.

[5] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. 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)*, pages 49–5, November 2000.

[6] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[7] C. Liu, A. Sivasubramaniam, and M. Kandemir. 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)*, pages 176–185, Madrid, Spain, February 2004.

[8] P. S. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 8th Annual Simulation Symposium*, Phoenix, Arizona, USA, April 1995.

[9] M. Modarressi, S. Hessabi, and M. Goudarzi. A Reconfigurable Cache Architecture for Object-Oriented Embedded Systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 959–962, May 2006.

[10] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer . *IEEE Micro*, 25(1):90–97, 2005.

[11] N. Nethercote and J. Seward. 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.

[12] S. P. VanderWiel and D. J. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. *IEEE Computer*, 30(7):23–30, 1997.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on January 15, 2010 at 10:28 from IEEE Xplore. Restrictions apply.