Dynamically Reconfigurable Cache Architecture Using Adaptive Block Allocation Policy

Milene B. Carvalho, Luís F. W. Góes, Carlos A. P. S. Martins
Computational and Digital Systems Laboratory (LSDC)
Pontifical Catholic University of Minas Gerais - Belo Horizonte, Brazil
milene@ieee.org, lfwgoes@yahoo.com.br, capsm@pucminas.br

Abstract

In this paper, we present a dynamically reconfigurable cache architecture using adaptive block allocation policy analyzed by means of simulation. Our main objectives are: to propose a reconfigurable cache architecture and to propose, implement and analyze the performance of an adaptive cache block allocation policy. First, we present a proposal of the reconfigurable cache architecture that can adapt according to the workload. Then we present our adaptive policy and do some performance tests comparing our cache architecture with some set associative configurations. In these tests, we use some traces from BYU Trace Distribution Center of SPEC 2000 Benchmark. Finally, we analyze the results based on some metrics like cache miss ratio, response time, etc.

1. Introduction

An ideal memory for a computer system would have an infinite size and be extremely fast, that is, access time equal to zero and infinite bandwidth, but generally resources are limited. Thus, caches are designed to create the illusion to the processor of a big and fast memory [5].

The design of a cache is an optimization problem, like any computer design. This optimization is mainly related with the maximization of the hit ratio and the minimization of the access time [10]. Considering the constraints involved in the problem and these desired aspects, cache designers proposed three well-known cache organizations: direct mapped cache, fully associative cache and set associative cache.

Each organization can be better for a specific workload, that is, a specific memory trace behavior. However, it is difficult to design a cache that has a high performance for all different workloads of a general purpose processor. Thus, the designers choose cache organization/configuration that has a good performance for the most part of workloads or for the most used ones. Nevertheless, the ideal design must be optimized for all workloads. As it is not possible, an alternative is to adapt the cache organization to the workload, reconfiguring the cache or a part of it dynamically according to the executed workload characteristics [1].

In this work, we present a cache optimization based in the associativity. So, our reconfigurable cache architecture allows changing the set associativity dynamically reconfiguring itself. To perform cache reconfiguration, we developed an adaptive cache block allocation. The reconfigurable cache architecture allows other policies to be implemented, but in this paper we will describe only an adaptive cache block allocation policy based on set miss/hit and number of accesses. To verify it, we used simulation, because it appears as a less expensive (cost) alternative than cache implementation in hardware. Moreover, simulation allows detailed measurements and flexible configurations. It is also possible to determine workloads with desired characteristics [12].

Our main objectives in this article are: to propose a dynamically reconfigurable cache architecture and to propose, implement and analyze the performance of an adaptive cache block allocation policy. Our main goals are: the proposal of a dynamically reconfigurable cache architecture; proposal, development and implementation of an adaptive cache block allocation policy.

2. Related Works

Some works deal with changes in cache memory after design, dynamically adapting the cache structure or organization according to the workload. In this paper, we present a reduced number of these works. Almost all of them use monitors [7] to get information from workloads execution. Thus, an algorithm that predicts the new cache configuration for a given workload uses this information.

Considering spatial locality, there are works that present changes in the line/block [13] and in fetch [11] size. These approaches use the inherent spatial locality of applications and the memory traffic decreases. Lots of papers describe an cache associativity decreasing to minimize the cache energy dissipation while maintaining

1-4244-0054-6/06/$20.00 ©2006 IEEE

high performance [9]. The Reactive-Associative Cache (r-a cache) [2] provides flexible associativity. It has two types of positions: direct-mapped and set-associativity, the latter has a higher hit latency than the former.

Our approach present some new ideas not found in these presented works, like different associativity between sets and reconfigurable associativity with the same access time for all sets and entries.

3. Reconfigurable Cache Architecture

Reconfigurable computing was being applied, especially in hardware, with reconfigurable devices, such as FPGAs (Field Programmable Gate Arrays), contain an array of computing elements whose behavior are determined by configuration bits [4]. Our group has been working on a reconfigurable cache that dynamically changes its behavior according to the executed workload [3] based on concepts of reconfigurable computing [4][11]. The goal of reconfigurable computing is to allow that a reconfigurable object has its structure changed to a nonpredicted state of its design time. It allows an object to adjust its behavior to a specific situation. So, this object becomes flexible, leading to a high performance compared to an object with a fixed behavior.

In our reconfigurable cache architecture (represented by Figure 1), the configuration of cache's behavior is determined by our adaptive cache block allocation policy. This policy has the parameters of system's workload and/or cache performance metrics as an input and chooses, from possible solutions (configurations), one that will configure the cache behavior.

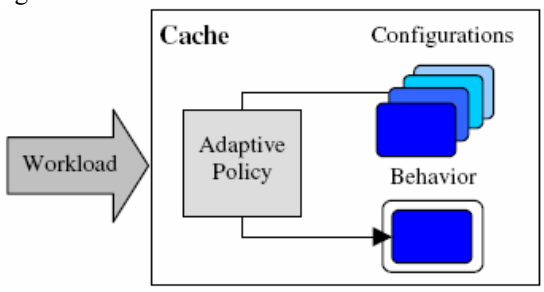

Figure 1. Reconfigurable cache architecture

In a *n*-way set associative cache, there are several sets, each one with exactly *n* entries. When the processor decodes a memory instruction and a requisition is sent to the cache, part of the address is used to locate the cache set that must be accessed. Then *n* set entries tags are verified. Our reconfigurable cache works like a set associative organization but it does not have the constraint of all sets with *n* entries. It has an initial and a maximum associativity. The latter indicates the maximum set entries that can be simultaneously verified. During the cache execution, it can dynamically adapt to the workload, changing the number of entries of each set. This number can change from one to the maximum associativity (number of comparators). In spite of these changes, the cache size is always the same of cache design. To determine the new number of entries to each set, an adaptive cache block allocation policy was designed.

4. Adaptive Block Allocation Policy

A block allocation policy assigns blocks to cache sets according to some parameters and restrictions. In our policy, we consider two parameters: the number of accesses and number of cache misses per set. The number of accesses indicates the importance level of a set and number of cache misses indicates if the number of blocks is sufficient to support the demand. These parameters are collected for a slice of time, defined as quantum. In a quantum the parameters are collected only for a processor task, so we can consider that there is one quantum for each task. It is done because each task has a behavior different of other tasks. During each quantum of a process, a statistical table is filled. This table contains performance metrics: the number of accesses and cache misses per set during the quantum and their means.

A number of accesses or cache misses is considered large if it is higher than or equal its respective mean of all the cache sets, otherwise it is considered small. An LL (large-large) set has a large number of misses and accesses, so we can consider that it is a highly accessed set with an insufficient number of block entries. Spite of having a small number of accesses, a LS (large-small) set has an insufficient number of block entries and can be improved. An SL (small-large) set has an ideal condition, because it is highly accessed and keeps a low cache miss ratio. A SS (small-small) set has a small number of accesses and misses, so it can give a block to a more important (large number of accesses) set or with a higher cache miss rate.

The three restrictions to assign blocks to sets considered in our policy are: maximum associativity value (number of comparators), number of cache sets and number of cache block entries. The increase of associativity in cache implementation really improves performance, doing a parallel search in a set to find a specific block. However, as the cost to add comparators is very expensive, designers must evaluate the cost and benefits of a higher number of comparators. As a consequence of a limited number of comparators, the number of blocks in a set cannot exceed the associativity number. On the other hand, this number cannot be less than one block, to maintain the number of sets in cache. We consider that the number of cache sets is fixed, because its variation can invalidate past configurations

(number of blocks per set), parameters (number of access and cache misses) generating re-allocation of blocks and to simplify our block allocation policy.

Based on statistics obtained during the last quantum of a specific task, our adaptive policy allocates the blocks among the cache sets to improve performance (reduce the cache miss rate). Our adaptive policy takes blocks of SS sets (donors) and gives to the LL and LS sets (receptors). Although both LL and LS sets can receive block entries, the LL sets have higher priority, because high miss rate in high accessed sets can have a higher performance cost than a less accessed set.

In the end of a quantum, our adaptive policy classifies all sets in one of the possible combinations. Then, it creates a queue of donors and receptors. In the receptors list, the LL sets come first followed by the LS sets. Then we match donors and receptors in a FCFS (first-come-first-served) way, until one of the queues becomes empty. We decrease in one the associativity of donor set and increase in one the associativity of receptor set and then both are removed from lists. Then, the cache is reconfigured and the workload is executed. This operation can be done many times (quanta) as needed to reach the end of a task. In each end of quantum, the reconfiguration is performed. Beyond of reconfiguration overhead, we must consider the overhead imposed by the adaptive policy. To determine the total overhead of reconfiguring a cache we have to consider some implementation choices as the architecture level on each the adaptive policy is implemented and the quantum size.

5. Experimental Results

To analyze the performance of our cache architecture using our adaptive block allocation policy, we developed a simulator implemented in Java and verified using known traces from Brigham Young University Trace Distribution Center [14] available on Internet. They have all memory references occurred in an execution, including operating system references.

In our simulations, we used only data access (read/write) from the available traces. The memory traces used were six traces collected from SPEC benchmarks [15] running on Pentium III and Windows 2000 from BYU Trace Distribution Center [6]. The SPEC benchmarks used in the simulations and the number of data memory accesses used were: 256.bzip (3746058), 186.crafty (3861895), 164.gzip (3647919), 254.gap (4058463), 197.parser (4099236) and 181.mcf (3874037).

In these simulations, the computer configuration is based on an Athlon XP 2000 cache and primary memory times. These times are: primary memory latency equal to 139ns; primary memory read time equals to 3.42ns; primary memory write time equals to 5.81ns; cache memory access time equals to 2ns; cache memory write time equals to 0.28ns and cache memory read time equals to 0.32ns. We used a 512MB primary memory, 64KB data cache memory, 32 bits word size and blocks with 4 words. Word and block size is always the same. The cache uses LRU (Less Recently Used) replacement policy and write-back strategy. 2-way and 4-way set associative caches were simulated for performance comparison. The reconfigurable cache architecture simulated has as initial associativity equal to 2 and 4 comparators, indicating the maximum associativity of 4. The cache size (quantity of blocks that can be stored) used in all simulated configurations is the same one. As the number of blocks that can be stored in a cache is the same in all simulations, the number of sets is variable. The sets number of the reconfigurable cache is equal to a 2-way set associative cache (initial condition).

For all simulations we considered a quantum of size equals to operational system quantum. Using this value, the overhead can be amortized, considering that the reconfiguration process is realized during the context switch, since the task processing has to be interrupted and some "new" task context must be loaded.

Our reconfigurable cache with adaptive policy has a miss ratio smaller than a 2-way set associative cache organization. As a reconfigurable cache has 4 comparators, it was expected. In 256.bzip and 164.gzip we found a difference between 2-way and 4-way set associative caches similar to the difference between 4-way set associative and reconfigurable caches (Figure 2).

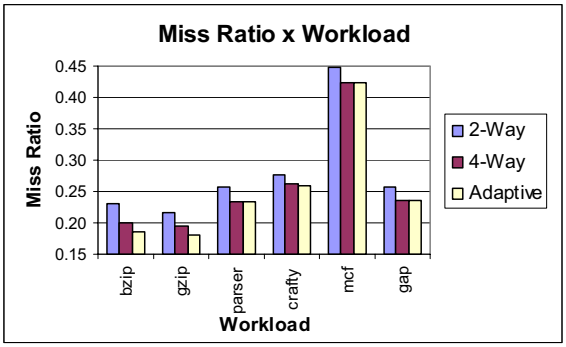

Figure 2. Miss ratio of executed workloads

In traces 197.parser and 186.crafty the difference between set associative 4-way and reconfigurable caches miss ratio is small (Figure 2). Our cache has a miss ratio 0.075% for 197.parser and 0.29% for 186.crafty smaller. In traces 181.mcf and 254.gap the difference between set associative 4-way and reconfigurable caches miss ratio is small. But our cache has a miss ratio higher than 4-way. As explained before, it is necessary to analyze this difference considering the cache memory area. This

difference could be allowed in a design that cares about used memory area as an example.

The adaptive policy works better on applications (workload) that access sets in a heterogeneous. So, the policy tries to balance the distribution of blocks, giving more blocks to sets more accessed and with more cache misses. Thus, the access conflicts in these sets are reduced. Workloads with different types of memory traces will enforce the policy to adapt the cache many times. The adaptive policy does not present good improvement with applications that access addresses in a homogeneous way. Because more accessed sets will steal blocks from less accessed but still high accessed sets. This will unbalance the distribution of blocks among the sets and can decrease performance.

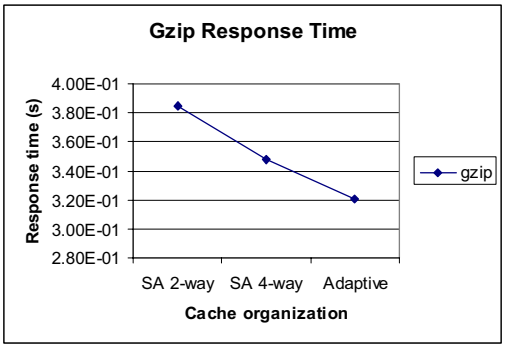

Figure 3. Response time of gzip trace

A miss ratio decreasing improves the workload response time (a lower response time). As an example, Figure 3 represents the 164.gzip response time for the simulated architectures.

6. Conclusions

In this work, we presented a reconfigurable cache architecture and an adaptive cache block allocation policy analyzed by means of simulation. So, we used a real computer (memories size and times) configuration to measure the response time and miss ratio from execution of real memory traces. We proposed, implemented and tested the adaptive policy using traces from BYU Trace Distribution Center (workload). Finally, we compared the execution of classic cache organizations (set associative) against our reconfigurable cache with the adaptive block allocation policy through some metrics. So, we concluded that the simple adaptive policy can find best cache configurations, decreasing significantly miss ratio and response time. In spite of this, the policy could work improperly on some applications that access memory sets in a homogeneous way. In this case, the policy could not improve the performance, but with other policies, the reconfigurable cache could reach better performance than a fixed cache.

Some improvements on our reconfigurable cache architecture and adaptive policy are: adaptation based on other cache parameters like block size etc; avoidance of adaptation on application with homogenous memory accesses.

Our main contributions described in this article are: the proposal of a dynamically reconfigurable cache architecture; proposal, development and implementation of an adaptive cache block allocation policy.

As future works, we remark: test and verification of multilevel caches; improvement of the adaptive policy; implementation of new adaptive policies.

7. References

[1] D. H, Albonesi. Et al. Dynamically tuning processor resources with adaptive processing. IEEE Computer, 12(36):49-58, 2003.

[2] B. Batson, T. N. Vijaykumar. Reactive-associative caches, Proceedings of IEEE International Conference on Parallel Architectures and Compilation Techniques, pp. 49-60, 2001.

[3] M. B. Carvalho, C. A. P. S. Martins. Cache Architecture with Reconfigurable Associativity, 5th WSCAD, pp. 50-57, 2004. (in Portuguese)

[4] K. Compton, S. Hauck. Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Survey, 34(2):171-210, 2002.

[5] J. L. Hennessy, D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufman, 3th Edition, 2003.

[6] J.L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium, IEEE Computer, 7(33): 28-35,2000.

[7] T. L. Johnson, D. A. Connors and W. W. Hwu. Runtime adaptive cache management, Proceedings of the 31th HICSS,7(6-9): 774-775, 1998.

[8] C. Martins, E. Ordonez, J. Corrêa and M. Carvalho. Reconfigurable Computing: concepts, tendencies and application. In: XXII JAI, SBC2003, Vol. 2, pp. 339 – 388 2003. (In Portuguese)

[9] M. D. Powell, A. Agarwall, T. N. Vijaykumar, B. Falsafi and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping, Proceedings of 34th ACM/IEEE International Symposium on Microarchitecture, pp. 54-65, 2001.

[10] A. J. Smith. Cache Memories, ACM Computing Surveys, 14(3):473-530, 1982.

[11] W. Tang, A. Veidenbaum, A. Nicolau and R. Gupta. Cache With Adaptive Fetch Size, Technical Report ICS-00-16 of University of California, Irvine, April 2000.

[12] R. A. Uhlig, T. N. Mudge. Trace-driven memory simulation: a survey, ACM Computing Surveys, 29(2):128-170, June 1997.

[13] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, X. Ji. Adapting Cache Line Size to Application Behavior, Proceedings of the 13th ACM ICS, pp. 145-154, 1999.

[14] Brigham Young University Trace Distribution Website: http://traces.byu.edu/

[15] SPEC - Standard Performance Evaluation Corporation Website: http://www.spec.org/