DYNORA: A New Caching Technique

Srivatsan P. Sudarshan P.B. Bhaskaran P.P.

Department of Electrical and Electronics Engineering
Sri Venkateswara College of Engineering, Chennai, India
e-mail: srivatsan00@yahoo.com

Abstract

Cache design for high performance computing requires the realization of two seemingly disjoint goals of higher hit ratios at reduced access times. Recent research advocates the use of "resizable" caches to exploit cache requirement variability in programs. Existing schemes for resizable caches effectively employ either of the two fundamentally different methods: by changing the cache organization itself or by using a proper resizing strategy, that is, either static or dynamic resizing. Our paper looks at a new dynamic resizing strategy that aims at run time manipulation of the cache parameters to improve its performance. Two algorithms for dynamic reconfiguration are proposed and the results explained.

1. Introduction

Increasing problem sizes in High Performance Computing require efficient caching methodologies. Current research is being focused on energy efficient cache architectures ([1],[3],[5],[9]) and new reconfigurable caching techniques ([2],[4],[8]). Since circuit level techniques are not able to single handedly provide solutions for achieving the above mentioned ends, higher levels of abstraction namely Algorithmic and Architectural levels [7] are being looked at with increasing interest.

Present caching techniques utilize only separate, dedicated, associative-memory hardware structures. This fixed division of fast memory into separate pieces cannot achieve efficient fast memory access across all applications. Also, they do not take advantage of the variations in problem size for the base design of the cache because of which the end performance is affected. With the advancements in architecture over the years, the pattern of memory referencing has also undergone changes. The incorporation of multithreaded parallel processing has resulted in patterns that have less temporal locality. Multitasking makes the problem worse by interleaving logically unrelated, but possibly interfering cache accesses. One way to reduce these problems is the implementation of the brute-force method of increasing the cache size or designing caches with higher associativities. But already caches are occupying a large fraction of the chip area. A better solution would be seeking of improved ways of cache utilization. Static Cache partitioning is an old idea wherein Instruction and Data caches have already been split in Harvard architectures. The main disadvantage with static partitioning is that it wastes resources by allocating either too much or too less.

Misses in caches can be classified into four categories: conflict, compulsory, capacity, and coherence [6]. Conflict misses are misses that would not occur if the cache was fully-associative and had LRU replacement. Compulsory misses will occur in any cache organization because they are the first references to an instruction or piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references. Coherence misses are a result of invalidation to preserve multiprocessor/multitasked cache consistency. Even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered. In fact, the direct-mapped cache is the only cache configuration where the critical path is merely the time required to access a RAM.

One way of reducing the number of capacity and compulsory misses is to use longer cache line sizes or using hardware prefetching methods. However, line sizes cannot be made arbitrarily large without increasing the miss rate and greatly increasing the amount of data to be transferred. Coherence misses can be effectively reduced by using a sound placement and replacement policy, wherein the interleaving of unrelated data is reduced.

In this paper we evaluate the opportunity for a dynamic resizing strategy (DYNORA) in cache memory architectures. We also propose two hybrid resizing techniques and compare them with the existing solution for recon-

Proceedings of the Euromicro Symposium on Digital System Design (DSD'03)
0-7695-2003-0/03 $17.00 © 2003 IEEE

figurable caches ([5],[2]) with emphasis on what types of cache misses our scheme will improve. The rest of the paper is organized as follows. Section 2 gives a brief overview of conventional performance improvement schemes for caches and also mentions recent trends in recent research. Section 3 describes DYNORA and aims at explaining our implementation of Performance-on-Demand caching. Simulation Results are explained in Section 4 and Section 5 gives a summary of the paper and describes future work.

2 Improving cache performance by reducing Cache misses

2.1 Conventional Schemes

Most cache research is concentrated on reducing the miss rate. Conceptually, conflicts are the easiest to handle: Fully associative placement avoids all conflict misses, but is very expensive to implement in hardware. Little can be done to capacity misses other than to enlarge the cache size. This reduces thrashing of data between the main memory and the cache, but since cache memory already occupies a large percentage of the chip area, this is not feasible always and the issue of power dissipation comes into the picture.

Importantly, changing one or more cache parameters for reducing a particular type of miss might have an impact on performance in case of another type of miss and on the access time of the cache itself. Thus any scheme to improve cache performance should try to achieve a balance between the various apparently conflicting factors.

Conventionally, the following six techniques are used to reduce the miss rate in caches. They are: having a larger block size, implementing higher associativity, using victim caches, hardware prefetching, compiler-controlled prefetching and by compiler optimizations. A detailed explanation of these is given in [6].

In addition to reducing miss rate, performance can also be improved by reducing the cache miss penalty. Perhaps the most interesting of the techniques for this has been the idea of adding a second level cache, in between the original cache and the main memory.

2.2 Recent Advances

Recent trends in improving performance have been focused mainly on reducing the power dissipation in caches without affecting its performance [1]. Research has also been done in trade-offs between the cache parameters. [3] gives a case study of cache design trade-offs for power and performance optimizations. Since [5], the focus of research has shifted to run-time reconfiguration methods with on-demand-performance as the keyword. This paper explored the concept of sub-array partitioning of set associative caches, both to conserve power and improve performance. In [2], an integrated architectural and circuit-level approach to reducing leakage energy in instruction caches is explored in detail. At the circuit-level, the DRI i-cache employs gated-V_{dd} to virtually eliminate leakage in the cache's unused sections.

The cache parameters can be set either dynamically or statically. As a first step towards understanding a dynamically resizable cache design, [2] focuses on designs that statically set the values for the parameters prior to the start of program execution. Though this paper looks at cache resizing, it is not totally dynamic. Here is where our work gains significance, as we propose a truly dynamic cache resizing strategy. We proceed to explain our DYNORA (DYnamic Need Oriented Resource Allocation) concept in the following sections.

3. DYnamic Need Oriented Resource Allocation

DYNORA is a new caching technique in which a set of parameters are used to monitor, react and adapt to changes in the application that is currently running the cache. Figure 1 shows a DYNORA cache and its important elements. We use the miss rate as the most important parameter for monitoring the cache performance.

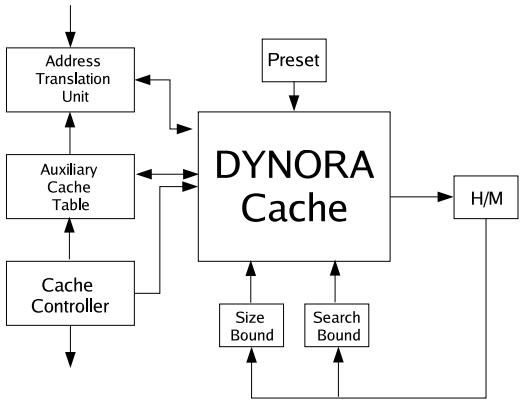

Figure 1. Anatomy of a DYNORA Cache

It is well known that the range of caches from direct mapped to fully associative is really a continuum of levels of set associativity: direct mapped is simply 1-way set associative and a fully associative cache with m blocks can be thought of as a m-way set associative cache. Alternatively, a direct mapped cache may be thought of as having m sets and a fully associative cache as having just a single set.

A DYNORA cache differs from a conventional cache in the sense that its associativity is not fixed; that is, it can vary in between a fixed upper and lower limit. The novelty is that the upper and lower limits can be preset. Moreover, in a DYNORA cache, the blocks belonging to a given set need not be adjacent as in a conventional cache; the set can contain 'n' blocks from any location (according to the current associativity). This is elucidated in fig. 2 .

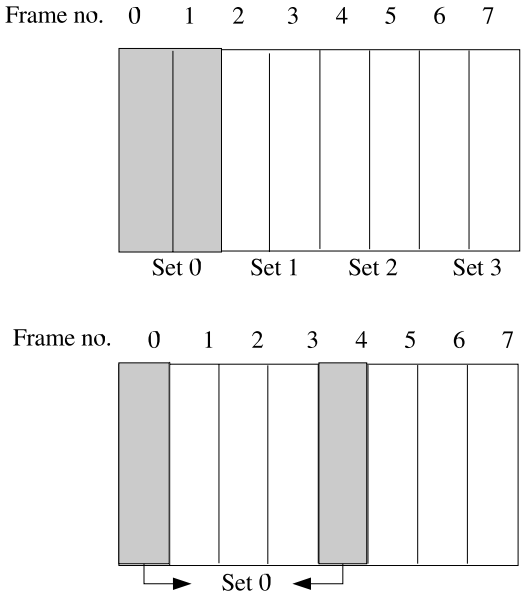

Figure 2. Conventional Cache Vs. DYNORA Cache

With respect to fig. 1, the initial or the starting cache associativity is preset. The Auxiliary Cache Table (ACT) maintains information about what is the current cache associativity and which blocks in the cache belongs to which set. We shall explain the significance of this in the forthcoming paragraphs. The cache is controlled by a Cache Controller (CC), which forms the interface between the processor and the cache. The subsystem also contains a search bound and a size bound that form an integral part in determining the cache associativity for the next run.

Having thus defined the features of a DYNORA cache, we now proceed to explain the actual caching process. With reference to fig. 1, the most important elements of the cache subsystem are: the cache controller and the auxiliary cache table. We now present two algorithms, DY-1 and DY-2 exploiting the new type of cache.

The algorithm for DY-1 is as follows:

Let
$nAss =$ Current Associativity
$nCSize =$ Cache Size
$nBSize =$ Block Size

Algorithm: DY-1 using DYNORA
begin
get $nAss, nCSize, nBSize$
get $Cache\ Address$
decode $Address$ to get tag and $index$
Start **Search** at $index$ for tag

if {Hit}
begin
if $(nAss \neq 2)$
$nAss = nAss/2$
end
else
retain ACT
if {Miss}
begin
Search N adjacent sets
if {Hit}
begin
Swap {Current Block, LRU}
if $(nAss \neq nCSize/(2 * nBSize))$
$nAss = nAss * 2$
end
if {Miss}
begin
if $(nAss \neq nCSize/(2 * nBSize))$
$nAss = nAss * 2$
end
end
end

DY-1 works as follows: We assume that currently a user program is running and we investigate the caching operation. Assume that 'k' denotes the current associativity and nCSize and nBSize denote the cache size and the block size respectively. Once the processor generates the address to be looked up in the cache, it is fed into the Address Translation Unit (ATU). The ATU then generates the tag and index for the cache lookup based on the data in the ACT (the ACT has information about current cache associativity and what blocks belongs to which set). Once the address is generated for lookup and the data is searched for, the Hit/Miss flag (H/M flag) is set.

Now we propose to reduce the associativity of the cache by a factor of 2 in case of a hit. But in case of a cache miss, we proceed to search N nearby blocks, N depending on the cache size. If the data is found in this search, we propose to

swap the block where the search is initiated and the block where the data is found in the corresponding sets (see fig 2). This is the major design feature of DYNORA. In case the search doesn't result in a hit, a cache miss has occurred and the main memory is referenced for the data. In case of such a search-and-miss we propose to increase the associativity of the cache by a factor of 2.

After every cache access, the ACT is updated by the Cache Controller and this information is signaled to the processor. Of course, this associativity increase/decrease operation must be done in such a way that the cache neither becomes fully associative or direct mapped. The search bound and the size bound units control the number of blocks to be searched and upper and lower limits of cache associativity, respectively.

For programs with large temporal incoherence, we propose another algorithm closely on the lines of DY-1, but with a more complex search and replace algorithm.

As seen above, although the associativity of the cache varies dynamically in DY-1, at any given time it is constant throughout the cache. Though it may change in the next cache access, all sections of the cache have the same associativity. In DY-2, it is proposed that different sections of the cache may have different associativities. Our novel idea that even non-adjacent frames can be part of a set gains importance in this context. If this feature were not available in a DYNORA cache, then it is impossible to conceptualize different sections of the cache having different associativities. In the following paragraphs, we present the DY-2 algorithm.

As mentioned previously, in DY-2, different sections of the cache have different associativities. The ACT, in addition to maintaining information about which block belong to which set, should also contain data about which portions of the cache have what associativities. In DY-2, in case of a cache hit, the current cache associativity is retained. If a cache miss occurs, searching is initiated in N adjacent blocks. If data is found in this search, instead of swapping (as in DY-1), the associativity of the current index where the search is initiated is increased so that it now includes N additional blocks. Once the associativity of the current set is altered, the blocks near it lose their associativity. These blocks are dynamically allocated to a new index with the current associativity. In case of a miss after the search, the associativity of the cache is increased by a factor of 2 as in DY-1. Here the block with the largest associativity is not adjusted; only the other sets are adjusted.

As can be seen, DY-2 is slightly more difficult to implement than DY-1. This algorithm, if used for normal programs (i.e. not in a multitasking environment) can lead to excessive and unheeded cache reconfigurations. This might lead to excessive power dissipation. Currently we are working at simulating the DY-2 algorithm.

Algorithm: DY-2 using DYNORA
begin
get $nAss, nCSize, nBSize$
get $Cache\ Address$
decode $Address$ to get tag and $index$
Start **Search** at $index$ for tag

if{Hit}
retain ACT
if{Miss}
begin
Start **Search** in N adjacent blocks
if{Hit}
begin
calculate $nAss + N$
Set $nAss = nAss + N$ for given index alone
Start partitioning of remaining blocks
begin
For all possible adj. blocks retain the current associativity.

Assign all non-adj. blocks for $nAss = k$ till all possible non-adj. blocks are assigned.
end
update ACT
end
end
if{Miss}
begin
Set $nAss = (k * 2)$ for blocks other than the block with largest associativity
update ACT
end
end

4. Simulation Results

We modeled a unified cache memory that had the features of DYNORA using the C++ programming language. We used LRU policy for data replacement and write back on a cache miss. The program computes the average memory access time based on the model given in [6]. The model was programmed such that it could also be run as a conventional cache. The program was written to handle the DY-1 caching algorithm. The simulations were done for various associativities for both conventional and DYNORA cache. In case of DYNORA cache, this would be the starting associativity for the program. The results of the tests are given in figures 3 and 4. We measured the difference in hit ratio and access time between a normal and DYNORA cache.

Referring to fig. 3, we see that there is definite increase in the hit ratio for a DYNORA cache. It can be seen that

the comparison is between a normal cache of fixed associativity and a DYNORA cache of varying associativity. It is important to note that the DYNORA cache will soon change its associativity, and the average associativity of the cache through the entire program, will definitely be more than that of a normal cache. Though superficially it might seem that the comparison is void, the significance of DYNORA comes out only because of this comparison. This is because DYNORA methodologies try to find out the optimum associativity for the cache, based on the type of program. It was found out that the associativity of a DYNORA cache does not change after a particular limit, and this is reflected in the graphs. The flat portion towards the end of both the graphs suggest that the percentage increase in hit ratio and access time become constant after a particular limit for all associativities shown.

The significance is that for achieving identical results with a normal cache, either the associativity or the block size or the overall cache size has to be increased hugely. This would be prohibitive from both the energy dissipation cost point of view. With DYNORA, it is possible to retain the cache and block sizes but still improve the parameters.

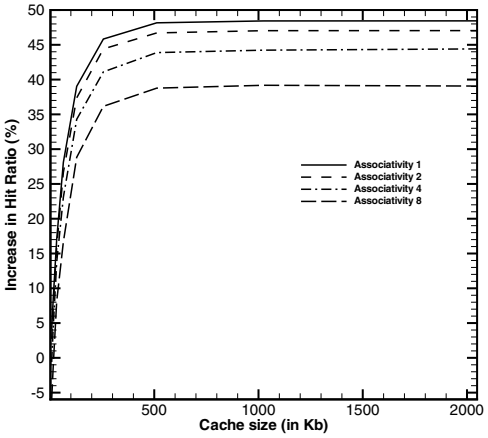

Figure 3. Impact on Hit Ratio

5. Conclusions

This paper presented two algorithms for run time reconfiguration of cache parameters using the concept of a Dynamic Set Associative Cache (DSAC). Simulated results for DSAC show very promising improvements in cache parameters. The implementation aspects, though not discussed, would not involve major changes in architecture. The discussed model will only require an improved cache controller and synchronizing schemes, which would be more economic than just increasing the cache size or designing caches with increased associativity. These aspects are being looked at, along with comprehensive simulations for the proposed DY-1 and DY-2 models. We expect the models to produce considerable power savings over conventional caching models. Since the cache power accounts for a large fraction of processor power dissipation, this outcome is significant. We are also working to benchmark the results of both the algorithms.

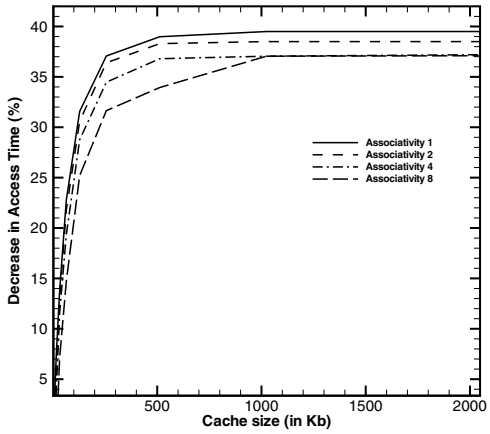

Figure 4. Impact on Access Time

References

[1] Falsafi. B., Vijaykumar. T.N., Roy. K., and Powell. M.D. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.

[2] Se-Hyun Yang, Powell. M.D., Falsafi B. and Vijaykumar. T.N. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[3] Su. C.L. and Despain. A.M. Cache design trade-offs for power and performance optimization: A case study. In *International Symposium on Low Power Design*, 1995.

[4] Chiou. D., Rudolph. L., Devda. S., and Ang. B.S. Dynamic cache partitioning via columnization. Memo 430, Computer Systems Group, Massachusetts Institute of Technology, 2000.

[5] D.H.Albonesi. Selective cache ways: On-demand resource allocation. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, pages 248–259, 2000.

[6] J. L. Henessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2nd edition, 1996.

[7] Inoue. K. *High-Performance Low power Cache Architectures*. PhD thesis, Konshu University, 2000.

[8] Zhang . M. and Asanovik. K. Highly associative caches for low power processors. In *Proceedings of 33rd International Symposium for Micro Architectures*, 2001.

[9] Ko. U., Balsara. P.T, and Mangione-Smith. W.H. Energy optimization for multilevel cache architectures for risc and cisc processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1998.