ATLANTIS PRESS
Advances in Computer Science Research, volume 44
3rd International Conference on Wireless Communication and Sensor Network (WCSN 2016)

A Cache Utility Monitor for Multi-core Processor

Juan Fang, Yan-Jin Cheng, Min Cai, Ze-Qing Chang
College of Computer Science, Beijing University of Technology
Beijing, China
Email: fangjuan@bjut.edu.cn

***Abstract*-In recent years, high performance computing systems have obtained more processing cores and shared a last level cache (LLC). Now, the problem to the existing cache partitioning techniques is that they give each core the number of cache ways according to their need, these schemes have the potential to realize significant performance increases, yet for most part they do not consider LLC energy saving. In this paper, we design and realize a multi-processing processor monitor. Through a utility monitor we calculate the number of hits and misses when allocate different cache ways to each application. In other words, we use utility monitors to track the access by each core to characterize each thread's use of the cache. Dynamically give each core the number of ways based on the performance to achieve its highest utilization. On gem5, we run Parsec benchmarks as our multi-threaded application. We output the numbers of misses for all possible number of ways, and find the number of associativity to achieve its highest utilization. By analysing experimental results, cache miss rate decreases with the increasing of the cache capacity.**

Keywords-Multicore; Shared cache; Cache partitioning.

I INTRODUCTION

With the popularization of multi-core and many-core technology, the operating system can greatly improve the processing capability and computing performance of the whole computer by parallelizing the computing tasks to different physical cores. The competition of shared resources introduced by multi-core technology, that is, the hardware resources of shared tasks such as caches and bandwidths running on various physical cores, has caused a certain degree of shared resource snatching, which makes the tasks on physical cores influence each other. And the overall computing performance and service quality of the system is degraded.

Shared cache effectively improve the utilization of bandwidth and cache space, but inevitably lead to the problem of resource competition. Generally, L2Cache adopts LRU (Least Recently Used) strategy, which effectively reduces the frequent page switching and memory access behavior, but exposes its own disadvantages in multi-core situation - there is a blind spot in the application which have different cache requirement and cannot distinguish effectively. Therefore, in the situation of shared L2Cache in multi-core resource competition, some of the poor local application may take up a lot of cache resources and generate frequent page switching. So, parallel applications cannot get enough cache resources and effective pages of competitive applications are constantly being replaced, resulting in lower cache hit rate, and serious performance loss.

Through the above analysis, we need to find a cache partitioning mechanism, which make effective cache partition according to the cache's demand of different applications, reduce the performance impact because of the cache resource competition, and improve computing performance.

Zhang L, Liu Y and Wang R, et al [1], proposes a malloc allocator-based dynamic cache partitioning mechanism with page coloring, to make page coloring-based cache partition more practical.

Khaitan S K and Mccalley J D[2] using decay cache technique, dynamically turn off the cache blocks to save cache energy.

Altmeyer S, Cucu-Grosjean L and Davis R I[3]investigate static probabilistic timing analysis (SPTA) for single processor real-time systems that use a cache with an evict-on-miss random replacement policy.

This article will discuss the design and implementation of cache utility monitor (UMON). In the second part, we will briefly describe the cache partitioning technology, that is, cache space prediction, cache allocation strategy and partition realization mechanism. In the third part, we will briefly describe the design and implementation of the utility monitor, the experimental process and the experimental results. The fourth part summarize this article and point out the direction of further research.

II CACHE PARTITIONING TECHNOLOGY

Cache partitioning is an effective means to eliminate shared cache interference, which controls on-chip cache space by software without the need to do too much cache structure changes. A cache partitioning scheme is divided into three parts: cache space prediction, cache allocation strategy and partition realization mechanism.

A. *Cache Space Prediction*

Before partitioning the shared cache space, we need to determine the cache allocation ratio. The LRU replacement strategy used in the modern cache structure does not guarantee the effective allocation of the shared LLC for each program. The LRU replacement policy is based on program memory requirements (Locality feature) to allocate cache space, which is the same for all programs.

There are many techniques for cache space allocation prediction, the most notably among them are SDP (Stack Distance Profile) and MRC (Miss Rate Curve). SDP will be discussed in the third part of this article. MRC is used to

describe the curve of cache miss rate when a program obtains different cache size, through control the (software or hardware) available cache space of program to get the corresponding cache miss rate. MRC can get the best program performance and cache space ratio.

B. *Cache Allocation Policy*

In the shared cache conditions, after predicting the cache space which programs take up, the following is to determine each program's cache allocation ratio according to specific indicators.

The utility function is designed according to the number of cache misses when the cache memory space in the a-way cache and the b-way cache(a < b)memory in the cache partitioning mechanism with the cache path granularity:

$$U_b^a = miss_a - miss_b \tag{1}$$

In order to determine the utility of programs, we need to add a dedicated utility monitor in the cache structure, dynamically obtain the cache misses and calculate the ratio of cache space allocation to achieve highest utilization.

C. *Cache Partitioning Mechanism*

Cache partitioning technology is classifying the access data of thread according to their locality and fixing the data in the specific cache interval according to the strength of data locality. Through this method of cache partitioning, the data utilization in the cache can improve, and reduce the cache miss rate.

The current cache partitioning mechanism can be classified into four categories: cache way partition, cache set partition, page partitioning, and the partitioning which controls cache insertion and substitution strategy.

III DESIGN AND IMPLEMENTATION OF CACHE UTILITY MONITOR

We deploy a cache utility monitor on each core to obtain contention thread's cache resourse usage. Then the collected information is passed to the partitioning algorithm to determine the cache capacity of each thread, dynamically shut down the unused cache way, to achieve static power savings.

A. *Stack Distance Profile*

The stack distance profile is a histogram for threads, which is under different cache resources, to analyze hits and misses of the data blocks, as shown in Figure 1.

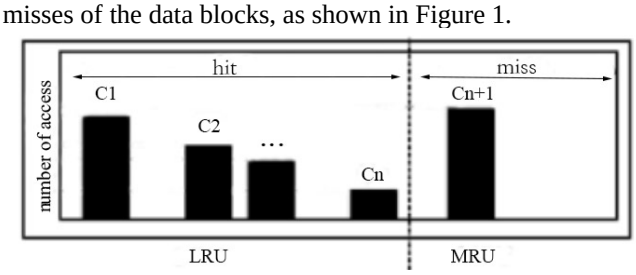

Figure 1. Histogram of stack distance profile

In the set-associative cache, the LRU replacement strategy presents a stack property, that is, in each set, cache block sort in order, the top of the stack store the most recently used (MRU) cache data block, and the least recently used (LRU) cache data block is stored in the bottom of the stack. The stack distance is the relative distance, which is from the top of the LRU stack to the access data block. LRU replacement strategy select the replacement and hits position according to the stack characteristics. If a cache miss, the data block pop which is in LRU position, the other positions data goes to the bottom of the stack, and the new data block pushes into the MRU position. If a cache block in the stack is hit, then the data block inserts into the MRU position, other data blocks whose distance is less than its stack distance turn down in order.

B. *Algorithm Description of Calculating Miss Number*

By using the stack property of LRU algorithm, we can calculate the number of L2cache misses generated by different associativity n '(n' <n) at the same time without changing the associativity of L2cache. For this reason, n + 1 counters are required for the n-way associative cache using the LRU replacement algorithm, C1, C2 …, Cn, C> n, each time the cache request, one of n +1 counter will plus one. If the request is a hit, assuming that the stack distance of the element is k, then the Ck counter is incremented by one. If the request is missing, the C> n counter is incremented by one.

Therefore, the number of misses generated by the n-way associative using the LRU algorithm can be expressed as:

$$misses_{n'} = C_{>n} + \sum_{i=n'+1}^{n} C_i \tag{2}$$

C. *Utility Monitor(UMON)*

LRU cache replacement strategy, which is multi-core processor using most, is an on-demand distribution method. It allocates the cache space according to required when program access to data, so cannot guarantee all programs on the shared last level cache obtain effective allocation of cache. Some strong local programs cannot obtain enough cache space cause performance degradation. Based on the utility of the cache allocation strategy, the program can effectively predict the miss rate in different associativity.

Each core is assigned a Utility Monitor. To monitor the application's usage efficiency, we need to track the number of cache misses generated by all possible associativity of cache. That is, we monitor the access of each core, describe the characteristics of each thread when using the cache, and use the SDP (Stack Distance Profile) to find the number of ways for each core to achieve its highest utilization. The results will be submitted to the partitioning algorithm. The architecture of cache monitor is shown in Figure 2.

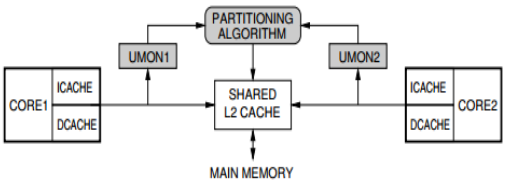

Figure 2. Architecture of utility monitor

D. *Cache Partitioning Based on Utility*

To determine cache partitions we use Algorithm 1. We are not concerned with allocating ways to each core individually, so we only need to find the number of ways for each core to achieve its highest utilization (max_mu). The mu is defined as follow. We assume all threads have equal priority and therefore give each core a number of ways based on the performance benefit it can realize from them. If thread were to have differing priorities we could incorporate this information into Algorithm1 by increasing the number of ways that high-priority threads receive and decreasing the number allocated to low-priority threads.

$$mu = \frac{getNumMisses(0) - getNumMisses(j)}{j} \quad (3)$$

IV EXPERIMENTAL RESULTS AND ANALYSIS

The data of cache characteristics, including cache miss rate, cache hit rate and cache reuse distance. Cache miss rate is the easiest to sample, so the cache miss rate is chosen as the experimental analysis data. In gem5[4], we run Parsec benchmarks[5] as our multi-threaded application. The application include blackscholes, bodytrack, canneal, ferret, fluidanimate, streamcluster and swaptions. We get the number of L2cache miss at stats.txt, through the calculation, we can get each Cache miss rate in the different associativity. At the same time, we can get the optimal number of ways for each core to achieve highest utilization. This data will be next used in cache partition decision.

A. *Miss Rate Under Different Benchmarks*

As is shown in Table 1 system configuration, we run seven benchmarks in gem5, and get different miss rate curve under different cache space.

TABLE I. EXPERIMENTAL CONFIGURATION PARAMETERS

Parameters	Configurations
Processor	4-cores
Shared L2Cache	4MB, 8-way

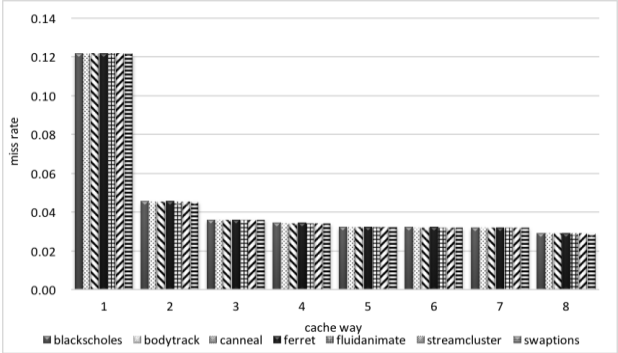

Figure 3. Miss rate in different benchmark

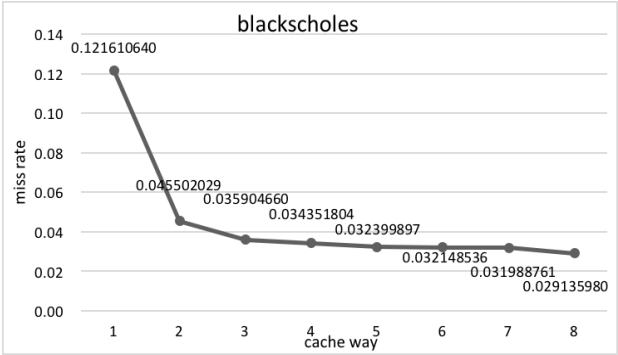

Figure 4. Miss rate of different associativity in blackscholes

According to the experimental results in Fig. 3 and Fig. 4, it can be concluded that with the increasing number of cache way, the miss rate is reduced. By comparing the data of seven benchmarks, it is found that the missing rate is basically the same, the difference is only after the decimal point. Bit, so we only analyze blackscholes.

B. *Miss Rate Under Different Numbers of Cores*

As is shown in Table 2 system configuration, we run the bodytrack / canneal benchmark in the gem5 to obtain the missing rate data under different cache spaces when the number of cores is 1, 2 and 4.

TABLE II. EXPERIMENTAL CONFIGURATION PARAMETERS

Parameters	Configuration
Processor	1/2/4-cores
Shared L2Cache	4MB, 8-way

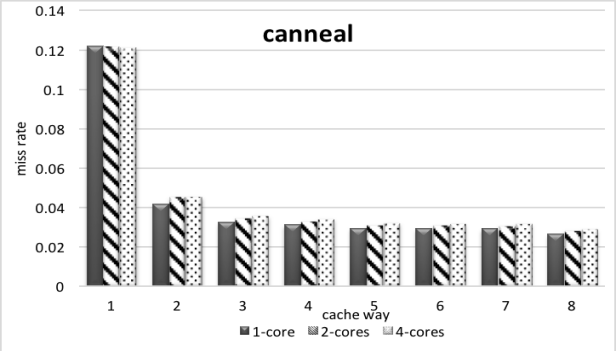

Figure 5. Miss rate of different number of cores in bodytrack

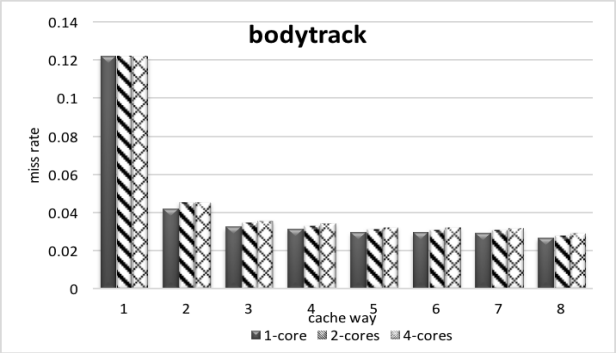

Figure 6. Miss rate of different number of cores in bodytrack

According to the experimental results in Fig. 5 and Fig. 6, we can see that the cache miss rate decreases with the increase of the cache capacity in different number of cores.

C. *Miss Rates at Different Cache Sizes*

TABLE III. EXPERIMENTAL CONFIGURATION PARAMETERS

Parameters	Configuration
Processor	4-cores
Shared L2Cache	512kB/1MB/2MB/4MB, 8-way

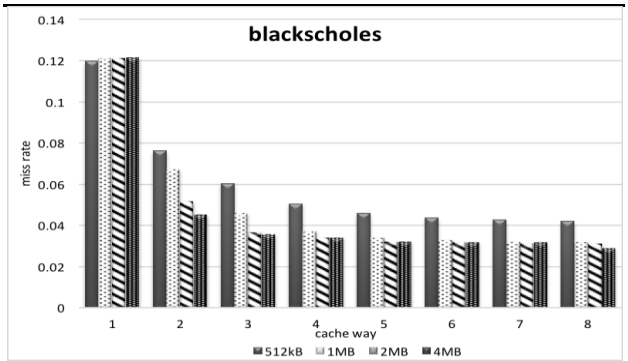

Figure 7. Miss rate of different size of LLC in blackscholes

By analyzing the data in Figure 7. As the L2 cache size increased from 512kB to 4MB, the missing rate is declining, when the associativity is greater than 1. And the same can be concluded. With the increase of L2cache associativity, the cache miss rate is decreasing.

In the experiment, we can obtain the number of cache ways for each core, so that they can reach the highest efficiency. This data will be used in the next step of cache partition decision, and finally achieve the goal of reducing dynamic and static power consumption.

V CONCLUSION

In this paper, we design and implement a utility monitor for multi-core processor. Through monitors to monitor each core access, we describe the characteristics of each thread using the cache, and use the Stack Distance Profile to determine how much program cache space could have better performance and dynamically allocate the best number of each of the core cache way, so that they achieve the highest utilization.

By observing the image of the cache miss rate obtained by the experiment, it can be concluded that the miss rate is decreasing with the increase of the cache associativity. We assign each core the best number cache way according to the Stack Distance Profile, this data will be used for future cache partition decision.

The latter part of the work will be divided data cache into shared area and private area, private data can only access the cache of private area, shared data can only access the cache shared area, according to the region-aware cache partition, dynamically shutdown the cache way when unused, and ultimately reduce the dynamic power consumption.

This research was supported by the Beijing Municipal Science and Technology Project (Grant No. Z151100002615032), along with other government sponsors. The authors would like to thank the reviewers for their efforts and for providing helpful suggestions that have led to several important improvements in our work. We would also like to thank all the teachers and students in our laboratory for helpful discussions.

REFERENCES

[1] Zhang, Ludan, et al. Lightweight dynamic partitioning for last-level cache of multicore processor on real system. *Journal of Supercomputing* **69**,33-38 (2014)

[2] Khaitan, Siddhartha Kumar, and J. D. Mccalley. Optimizing cache energy efficiency in multicore power system simulations. *Energy Systems* **5**,63-177 (2014)

[3] Altmeyer, Sebastian, L. Cucu-Grosjean, and R. I. Davis. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Systems* **51**,7-123 (2015)

[4] N. Binkert, B.M. Beckmann, G. Black, S.K. Reinhardt et al. The GEM5 simulator. *ACM SIGARCH Computer Architecture News* **39**, 1-7 (2011)

[5] Bienia, Christian. Benchmarking modern multiprocessors. *Dissertations & Theses - Gradworks* (2011).

Algorithm 1: Determining cache requirements.

```
Tways = N; /*Total number of cache ways in LLC */
        blocks_req[c] = 0; /*For each competing core, c */
        for each core c do
max_mu[c] = get_max_mu(c, Tways);
blocks_req[c] = min blocks to get max_mu[c] for core c;
        end
        return block_req;
get_max_mu(c, Tways);
max_mu = 0;
for j = 1; j <= Tways; j++ do
        U = change in misses for core c when moving from 0 to j ways;
        mu = U / j;
        if mu > max_mu then
max_mu = mu;
        end
end
return max_mu;
```