A Low-Latency Fine-Grained Dynamic Shared Cache Management Scheme for Chip Multi-Processor*

Jinbo Xu, Weixia Xu
College of Computer
National University of Defense Technology
Changsha 410073, China
{xujinbo, xuweixia}@nudt.edu.cn

Zhengbin Pang
College of Computer
National University of Defense Technology
Changsha 410073, China

Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha 410073, China

zhengbinpang@nudt.edu.cn

Abstract—**In order to utilize the shared last-level cache (LLC) in chip multi-processors (CMP) more efficiently, the partitioning of LLC resources among all cores should have the characteristics of low-latency for access, fine granularity for migration and simple hardware complexity for implementation. This paper proposes a dynamic LLC management scheme to achieve these goals. The proposed scheme migrates cache resources among different cores at the granularity of cache blocks, instead of ways. The quantity of victim cache blocks that each victim core can migrate to other target cores are related to an eviction probability, which are calculated according to the performance goal. Then the victim cache blocks for a target core is chosen from the nearest victim core who has non-zero eviction probability by introducing innovate E-Table structure in CMP. The eviction probabilities are updated periodically. With the help of E-Tables, the proposal achieves low-latency accesses by always keeping the required cache blocks near to the target cores. And fine granularity is guaranteed by maintaining an eviction probability for each core. In addition, only little additional hardware changes to traditional cache structure is required. Simulation results suggest significant performance improvements from 6.8% to 22.7% over related works.**

Keywords—cache management; chip multi-processor; low latency; fine granularity

I. Introduction

Research on last-level cache (LLC) management in chip multi-processor (CMP) pays more attention to fine granularity, low latency and simple implementation in recent years[1][2][3][4][5][6][7][8][9][10][11][12]. Finer granularity brings more flexibility to guarantee timely repartitioning. And lower latency makes cache accesses more efficient. At the same time, simple implementation reduces cost. Previous works on cache partitioning can be classified into coarse-grained partitioning[6][8][9][13] and fine-grained partitioning[10][14]. As for coarse-grained partitioning, way-partitioning is popular because of its simplicity of design. However, it can be inefficient as it only allows partition sizes to grow or shrink by a fixed large size (inversely proportional to associativity) while it is possible that the optimal size for a partition falls in between. As the number of cores increases and becomes comparable to the number of ways, such scenarios are likely to occur more frequently. Therefore, cache partitioning at finer granularity such as block level is more desirable. Vantage[10] achieves block-level cache partitioning, but it is only for a portion but not for all of the cache. In addition, the implementation cost is pretty large because of significant changes to the hardware. Manikantan et al. proposes PriSM[14], which manages the cache occupancy of different cores at cache block granularity by dynamically controlling their eviction probabilities. Moreover, PriSM requires only simple hardware changes to implement. However, the selection of victim blocks only relies on eviction probabilities, which may cause large access latency if the cache block is far from the corresponding core. To optimize the overall access latency, a good cache partitioning scheme should guarantee that cache blocks required by a core are always near to this core.

This paper proposes a shared cache management scheme for CMP, which incorporates the location distribution with the eviction probability, to achieve low-latency for access, fine granularity for migration and simple hardware complexity for implementation. The proposed scheme migrates cache resources among different cores at the granularity of cache blocks, instead of ways. At first, the performance goal is translated into eviction probabilities of each core to determine the quantity of victim cache blocks that each core could provide. Then, a victim core which is near to the target core and has higher eviction probability is chosen to provide the victim cache block for replacement. The eviction probabilities are updated periodically. The main contributions of this work show as follows.

Supported by the National Natural Science Foundation of China under Grant No.61202126, and the National High-Tech Research and Development Plan of China under Grant No.2012AA01A301, and the National Grand Fundamental Research 973 Program of China under Grant No.2011CB309705-1, and the National High-Tech Research and Development Plan of China under Grant No. 2013AA01A208

- In this work, each core maintains a look-up table, denoted as E-Table, to record the occupancy distribution of its surrounded cache resources. By using these E-Tables, the proposal achieves low-latency accesses by always keeping the required cache blocks near to the target cores.
- Fine granularity is guaranteed by maintaining an eviction probability for each core. Unlike way-partitioning where the victim core is identified based on the number of ways it occupies in the set, this work generates a victim-core id in line with the eviction probability distribution computed by the allocation policy. It also allows each cache set to make its own decision and controls the overall cache occupancy of each core at the whole cache level.
- Only little additional hardware changes to traditional cache structure is required. Simulation results suggest significant performance improvements from 6.8% to 22.7% over related works.

The remainder of this paper is organized as follows. In Section 1, the motivation is introduced. Section 2 presents the proposed cache management scheme in detail. The experimental results are described and analyzed in Section 3. Section 4 remarks the conclusion.

II. Motivation

The adjustment step of cache resources with coarse-grained cache partitioning scheme is too large. Consequently, theoretical optimal boundaries of cache resources among different cores can hardly be achieved. For instance, way-partitioning increases the allocation at a coarser granularity of one way in all the cache sets. In a 16 way cache allocating one more way translates to providing 6.25% more cache space. And in the case of the 64 and 256 way associative caches allocating one more way translates to increasing the occupancy by 1.6% and 0.39% respectively. As the cache size remains unmodified, the high associative configurations help us mimic the impact of partitioning at finer granularities. Increasing associativity and the resultant finer-grained control over cache occupancy helps improve the performance[15]. But building caches of very high associativity entails additional hardware and different cache organizations. Hence it is required to enable partitioning at granularities finer than that of associativity, such as block-level granularity. Block level cache partitioning provides the ability to reduce/increase the space allocated in steps of $1=N$, where N is the number of cache blocks. This paper will focus on the research of block level cache management scheme to achieve finer granularities, lower latency and simpler implementation.

Allocation and partitioning of shared cache resources are the key issues of block-level cache management scheme. The allocation method should guarantee that every bit of cache resources is owned by one of the cores and no idle cache block exists. Vantage[10] has the disadvantage at this point since it logically partitions the cache into 'managed' and 'unmanaged' regions and only achieves the desired target occupancy in the 'managed' portion of the cache by borrowing space from the unmanaged region, which inevitably wastes the unmanaged cache resources. Once all cache resources are allocated to all cores initially, partitioning scheme is required to dynamically transfer cache resources among different cores to achieve certain performance goals. Some questions should be answered for this transferring procedure, such as, which cache blocks should be transferred (i.e. the selection of victim caches), and how many victim cache blocks should be selected and when will them be transferred, et al. To answer these questions, the performance goals should be translated into the required cache resources for each core to determine the difference between the current cache utilization and the required cache utilization. These differences reflect the eviction probabilities of cache resources for each core, since a core who owns more free cache resources has more probability to provide victim cache for other cores. Therefore, eviction probability distribution can be considered as a selection criterion of victim caches. Secondly, the selection of victim caches should also consider the access latency to achieve more performance. Therefore, the proposal aims to achieving low-latency accesses by always keeping the required cache blocks near to the target cores, and achieving fine granularity by maintaining an eviction probability for each core. In addition, only little additional hardware changes to traditional cache structure is required.

III. The Proposed Cache Management Scheme

The details of the proposed cache management scheme are described in this section. For the convenience of description, TABLE I. defines some common terms used in this paper. N denotes the total number of cache blocks. W represents an interval of execution, measured in terms of misses in the shared cache. The allocation policy re-computes the cache space allocated to each core at the end of every interval. C_i is the ratio of the cache blocks occupied by Core$_i$ to the total number of cache blocks. M_i denotes the fraction of misses contributed by Core$_i$ to the total misses in an interval. T_i is the expected occupancy for Core$_i$ to meet a certain performance goal. τ_i represents the occupancy likely to be achieved at the end of an interval. E_i is the eviction probability of choosing a victim block from Core$_i$.

A. Fine-Grained Cache Partitioning Based on Eviction Probability

In this work, fine granularity is guaranteed by maintaining an eviction probability for each core. Unlike way-partitioning

TABLE I. Definition of common terms

Terms	Description
N	Number of cache blocks
W	Interval length. Measured in terms of number of cache misses.
C_i	Fraction of cache space occupied by Core$_i$ at the beginning of interval.
M_i	Fraction of Misses caused by Core$_i$ at the beginning of interval.
T_i	Desired cache occupancy at the end of interval for Core$_i$.
τ_i	Achieved target cache occupancy of Core$_i$ at the end of interval.
E_i	Eviction probability for Core$_i$.

where the victim core is identified based on the number of ways it occupies in the set, this work generates a victim-core id in line with the eviction probability distribution computed by the allocation policy. It also allows each cache set to make its own decision and controls the overall cache occupancy of each core at the whole cache level.

Eviction probabilities of all cores are calculated at the end of every interval of W misses. We assume that Core$_i$ accounts for a fraction M_i of the total misses in a certain interval. Therefore, the number of cache misses for Core$_i$ is $M_i \times W$. If no cache line belonging to Core$_i$ is evicted, its cache occupancy would have increased from C_i to $(C_i+(M_i \times W/N))$ during the interval. However, if eviction probability E_i is considered for Core$_i$, $E_i \times W$ lines would have been evicted over the interval. Therefore the cache occupancy of Core$_i$ at the end of the interval becomes $\tau_i= C_i+((M_i-E_i)\times W/N)$. The objective of the cache management scheme is to make τ_i reach the desired cache occupancy T_i as quickly as possible. If τ_i is not able to reach T_i in the current interval, E_i should be set to 0 when $C_i<T_i$(i.e., no cache block from Core$_i$ is evicted) and E_i should be set to 1 when $C_i>T_i$(i.e., cache blocks from Core$_i$ have the highest priority to be evicted). On the other hand, if τ_i can reach T_i in the current interval, E_i can be computed by solving the equation $T_i=\tau_i= C_i+((M_i-E_i)\times W/N)$. In summary E_i is calculated in every interval by using the following equation.

$$E_i = \begin{cases} 0, if((C_i - T_i)\times N/W + M_i) < 0, \\ 1, if((C_i - T_i)\times N/W + M_i) > 0, \\ (C_i - T_i)\times N/W + M_i, otherwise. \end{cases} \quad (1)$$

According to (1), C_i, M_i and T_i need to be determined first before calculating E_i. C_i and M_i can be obtained easily by deploying counters per core to provide the number of cache blocks occupied by each core and the number of misses during the previous interval. T_i is related to the performance goal and is computed with well-designed algorithm in this work. This paper focuses on Hit Maximization performance goal.

Hit Maximization tries to provide more cache space to the core that has the maximum potential to gain hits. Usually, the dependency upon cache resources of a certain core can be measured based on how well it is likely to perform if the whole cache were assigned to it. The core who has more dependency upon cache resources will be assigned more cache blocks. Therefore, the expected cache occupancy of Core$_i$, T_i, can be computed according to this theory. After this, E_i is determined by using (1). The computing of T_i and E_i is shown in Algorithm I.

If a certain core needs to be assigned more cache resources, a victim core is selected first from the cores which have non-zero eviction probabilities. Then a victim block belonging to the victim core selected in the first step is identified using the underlying cache replacement policy.

As for the core-selection step, multiple strategies can be used. For example, the core who has the highest eviction probability can be selected, or the victim core is selected randomly from the eviction probability distribution. And as for the block-selection step, any existing cache replacement

ALGORITHM I. COMPUTING OF TI AND EI

```
TotalProfit = 0;
TotalT = 0;
/*Dependency upon cache resources of a core is determined by
calculating the hit difference between stand-alone mode and shared
mode */
for (i = 1; i <= N; i++) begin
        Profit[i] = StandAloneHits[i] – SharedHits[i];
        TotalProfit += Profit[i];
end
/*Calculate the expected cache resources of each c=ore */
for (i = 1; i <= N; i++) begin
        T[i] = C[i] × (1 + (Profit[i] ÷ TotalProfit));
        TotalT += T[i];
end
/*Normalize T_i*/
for (i = 1; i <= N; i++)
        T[i] = T[i] ÷ TotalT;
Compute E_i using (1);
```

policies, such as LRU, DIP[16], TA-DIP[4], RRIP[5], RE-LIFO[1], can be used.

The above work achieves fine granularity by introducing eviction probability for cache partitioning, but access latency from core to cache is not fully optimized yet. The victim cache that the above work selects from multiple candidates may not be the nearest to the target core, which consequently loses performance. Therefore, in the next section, this paper tries to incorporate location distribution of cores and cache resources into the cache management scheme for reducing cache access latency.

B. Cache Partitioning Optimization for Low Access Latency Based on Location Distribution

The layout of cores and cache resources in CMPs varies a lot. For example, dance-hall layout[19] puts cores and L1 private cache around last-level cache, and Nahalal layout[20] puts last-level private cache around cores and L1 private cache, and last-level shared cache is surrounded by cores and L1 private cache. These two layouts have the disadvantages of poor scalability, which may cause long access latency from cores to cache resources. Tile-structured layout[21][22] is then widely used in CMPs because of its simplicity and scalability. Therefore, the proposed latency optimization strategy is based on tile-structured layout.

The basic structure of tile-structured CMP is illustrated in Fig. 1. Each tile contains one or multiple cores, private L1I/D cache, shared L2 cache, directory structure for cache coherence and a router for interconnection. Since L2 cache is shared, L2 cache located in a certain tile may be accessed by cores in other tiles. To make the cache access latency as low as possible, the network distance between each core and its corresponding cache resources should be as near as possible.

The basic idea of fine-grained cache management scheme based on eviction probability is to select a victim block from cache resources which are governed by cores who have non-zero eviction probabilities and then transform the victim blocks to the target core. If the victim blocks are selected without considering latency optimization, they can be selected randomly. However, this method would influences the overall

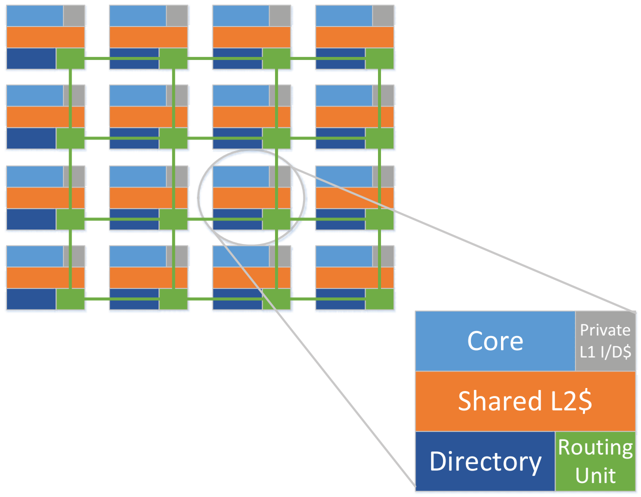


Fig. 1. Basic structure of tile-structured CMP

performance. To optimize the cache access latency, the selection of victim blocks should be guided with additional methods.

This paper incorporates location distribution of cores and cache resources into the cache management scheme for reducing cache access latency. Each tile maintains a look-up table, denoted as E-Table. E-Table in a certain tile stores all core IDs who own part of cache resources in the current tile. The eviction probabilities of these cores are also stored in the E-Table. In a tiled CMP, Tile(i, j) denotes the tile located in the i^{th} row and the j^{th} column, and Core(i, j) denotes the core in this tile. When Core(i, j) needs to get more cache resources from other cores, it is clear that the best candidates should be the cache resources in the current tile. To achieve this, the victim core is selected from the E-Table instead of from the entire CMP, since the cores who can be found in the E-Table own part of cache resources in the Tile(i, j). Therefore, if a core can be identified in the E-Table and the corresponding eviction probability is non-zero, this core is a candidate of the victim core. The selection among these candidates can be random, or the one with the highest eviction probability is selected. After this, a victim block is identified from the part of cache resources which belongs to the victim core in Tile(i, j) by using the underlying cache replacement policy. If at least one victim block can be found in Tile(i, j) for replacement, this victim block is reassigned to Core(i, j) from the victim core. Otherwise, if no victim block can be found in Tile(i, j), the search scope is expanded to its neighboring tiles. E-Tables in the neighboring Tile(i±1, j±1) are searched to find the victim cores. As long as the required victim blocks are acquired, the expansion of search scope stops. Fig. 2 illustrates the proposed cache partitioning optimization for low access latency based on E-Table. In Fig. 2, cache resources in Tile(i, j) are currently used by Core(i, j), Core(i-1, j), Core(i-1, j+1) and Core(i, j+1). Therefore, the IDs of Core(i-1, j), Core(i-1, j+1), Core(i, j+1) and their eviction probabilities are stored in the E-Table of Tile(i, j). By searching this E-Table, a victim core can be selected from these cores with non-zero eviction probability, for example, Core(i, j+1). Then victim blocks are selected with existing replacement policy(such as DIP[16]) from cache resources owned by Core(i, j+1) in Tile(i, j). The selection of victim blocks will not expand to other tiles unless no victim block is found in all cache resources in Tile(i, j). In this way, most of cache accesses of a certain core can be kept in the current tile or the neighboring tiles, which guarantees the spatial locality among each core and its corresponding cache resources, and therefore reduces the overall cache access latency.

C. *Low Cost Consideration about Hardware Implementation of the Proposed Method*

For the hardware implementation of the proposed method, only little additional hardware changes to traditional cache structure is required. In practice all cache accesses need to be tagged with the unique-ID of the core causing the access so that the cache-controller is able to keep track of occupancy. Since these requirements are common to all the cache partitioning/management schemes, the implementation costs of these requirements are not considered as additional costs of the proposed method.

As for the computation of eviction probability, only some hardware counters need to be deployed to provide the number of cache blocks occupied by each core and the number of misses during the previous interval for computing C_i and M_i. These counters are either already present in modern processors or proposed in earlier works[17]. In addition, it is possible to round off *TotalProfit* to the nearest power of 2 and convert the division to a shift operation. In this way, E_i and T_i can be computed using addition and shift operations on integers as long as N and W is a power of two. In Algorithm I, the total computations(arithmetic operations) performed by this algorithm ranges from 20 for 4-cores to 160 for 32-cores.

In each tile, the E-Table needs some additional space to store the IDs and the corresponding eviction probability values of the cores who has part of cache resources in the current tile. To reduce the hardware cost, this work stores the eviction probability values as integers. Experimental results show that the performance with 6, 8, 10 and 12 bits is very similar to that of using floating point to represent probability. Hence it is enough to use only 6 or 8 bits for eviction probability.

IV. Experimental Results

M5 simulator[18] is used in this work to evaluate the proposed cache management scheme. PriSM[14], Vantage[10], UCP[8], and PIPP[12] are studied together with our proposal for comparison. The configuration parameters of the simulator are listed in TABLE II. Core numbers of 4/8/16/32 are simulated. Cache line size is 64B. L1 cache has 64KB and is configured as 2 way. L2 cache is the last-level cache. For 4-core and 8-core system, L2 cache has 4MB and is configured as 16 way. For 16-core system, L2 cache has 8MB and is configured as 32 way. For 32-core system, L2 cache has 16MB and is configured as 64 way. This paper uses a set of multi-programmed workloads to evaluate the proposed method, including 71 workloads: 21 4-core workloads, 16 8-core workloads, 20 16-core workloads and 14 32-core workloads. Detailed simulation is carried out until all the programs execute

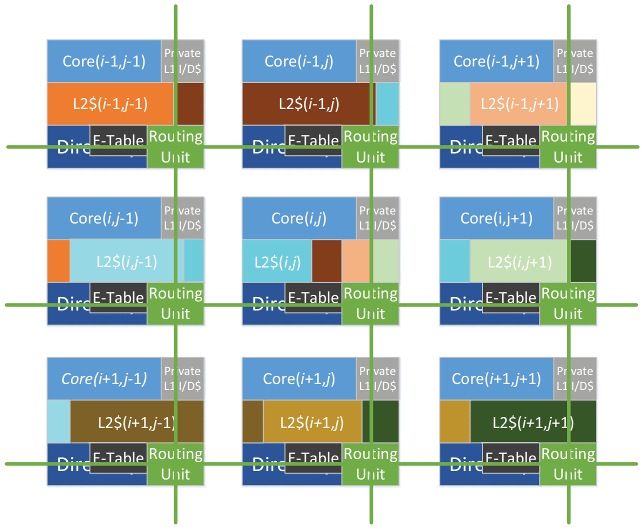


Fig. 2. Illustration of cache partitioning optimization based on E-Table

200M instructions. The experimental methodology and the number of instructions simulated are in line with earlier works[8][10][12][14]. We use Average Normalized Turnaround Time (ANTT)[15] to summarize performance. ANTT is defined as

$$ANTT = \sum (IPC_i^{SP} / IPC_i^{MP}) / n \qquad (2)$$

where IPC_i^{SP} is the stand-alone IPC of program in Core$_i$ and IPC_i^{MP} is its IPC when run as a part of the workload. ANTT is a lower-is-better metric.

A. *Performance Comparison with PriSM, UCP and PIPP*

Fig. 3 gives the results of performance comparison with PriSM, UCP and PIPP in terms of ANTT (normalized to that of LRU). It can be seen that our proposal achieves an average gain of 19%, 16%, 22.7% and 15% over LRU in the case of 4, 8, 16 and 32 cores respectively. Also our work outperforms UCP and PIPP at higher core counts. This work achieves 4.9% and 6.8% gain over PriSM for 16 and 32 cores scenarios. The main reason why our proposal outperforms related works includes:

TABLE II. CONFIGURATION PARAMETERS OF M5

Parameter	Value
Issue	4
Clock Frequency	4GHz
ROB/IssueQueue/LQ/SQ	96/32/32/32
L1 D/I Cache	64KB，64B Cache Line，2 way
L2 Cache (LLC)	4MB/8MB/16MB，64B Cache Line，16/32/64 way
Number of Cores	4/8/16/32
Memory Controllers	1/2/4/8
Main Memory Latency	50 cycles

- Fine-grained cache partitioning based on eviction probability allows each cache set to make its own decision so as to fully exploit the potential of every single cache block to gain maximal performance.
- Cache partitioning optimization for low access latency based on location distribution tracks the relative spatial information between cores and cache blocks to always keep the required cache blocks near to the target cores, therefore reduces the cache access time of each core.

It can be seen in Fig. 3 that our proposal achieves more performance improvements for systems with higher core numbers. The reason is that our proposal only selects victim cores from a subset of all cores instead of the entire set of cores. The higher the core number is, the smaller the proportion of candidate victim cores to all cores is. Therefore the cache resources are more convergent to their corresponding target cores, which consequently achieves more latency optimization.

Fig. 4 shows the detailed performance comparison of this work with PriSM, UCP and PIPP by using 4-core and 32-core workloads. The results suggest that our work outperforms related works for most workloads, such as Q7, Q11, T5, T7, T8.

B. *Implementation Cost Analysis of Locating Victim Caches*

The selection and locating of victim cache blocks directly influences the overall performance of the proposed cache partitioning scheme. If the victim blocks can be found in the local tile, high locality and low latency can be clearly guaranteed. Otherwise, if the victim blocks fail to be determined in a single iteration, more iterations are required to locate the victim blocks by searching the current E-Table for the other cores in the current tile. If failures happen again, the E-Tables in the neighboring tiles are then searched. More iterations mean more additional costs and less locality. Therefore, the number of iterations should be analyzed to evaluate the implementation costs.

This paper evaluates the implementation costs by using the ratio of the number that E-Table items are accessed to the

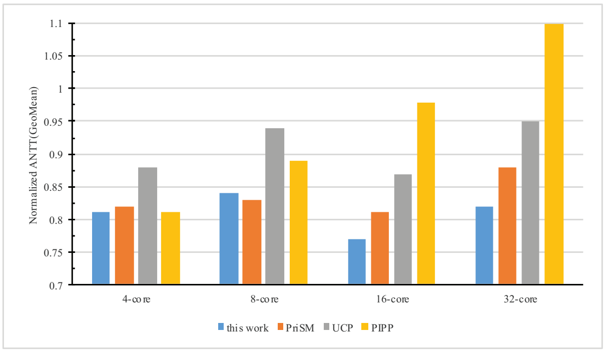
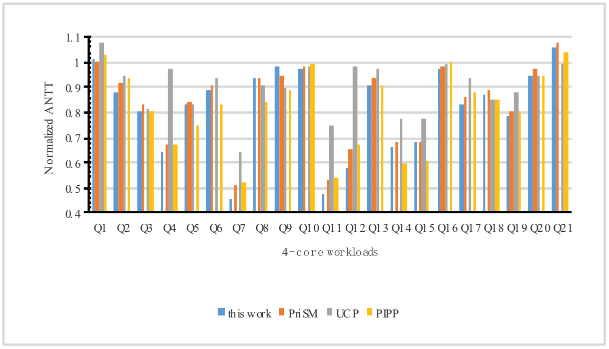
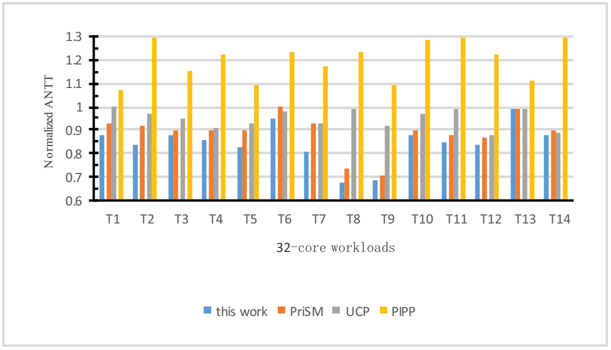


Fig. 3. Performance comparison with PriSM, UCP and PIPP



(a) Results of 4-core system



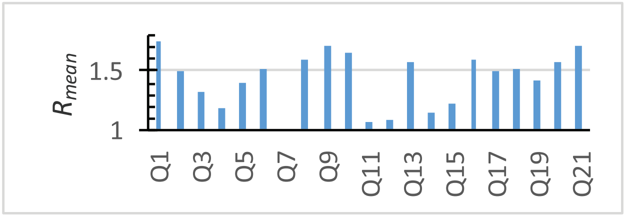
(b) Results of 32-core system

Fig. 4. Detailed performance comparison with PriSM, UCP and PIPP for 4-core and 32-core system

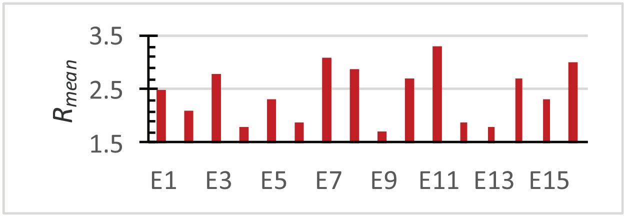
number of cache misses. This ratio is defined as R. The number that E-Table items are accessed can be retrieved by deploying hardware counters in each E-Table. Fig. 5 gives the mean values of R for all cores in 4/8/16/32-core system during 200M instructions. It can be seen that R value grows a little while the number of cores grows. Although the proposed method introduces some additional implementation costs, the cache access latency are also reduced significantly because of the improved spatial locality. Therefore, the proposed method still outperforms related works despite this additional implementation costs, as described in the previous section.

V. Conclusions

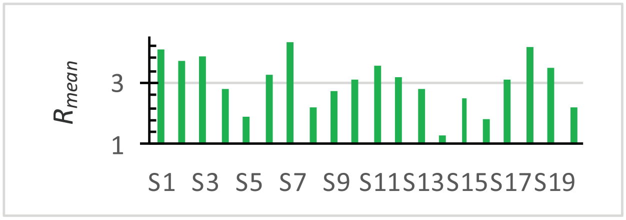
This paper proposed a low-latency fine-grained dynamic shared cache management scheme for chip multi-processor, which proved to outperform related works. The proposed scheme migrates cache resources among different cores at the granularity of cache blocks, instead of ways. Fine granularity is guaranteed by maintaining an eviction probability for each core. Furthermore, to optimize the cache partitioning scheme for lower cache access latency, this work introduced innovate E-Table structure in CMP to record the location distribution of caches and cores. With the help of E-Tables, the proposal achieves low-latency accesses by always keeping the required cache blocks near to the target cores. This paper also considered reducing hardware costs of the proposal. Only little additional hardware changes to traditional cache structure is required. Simulation results suggest significant performance improvements over related works.



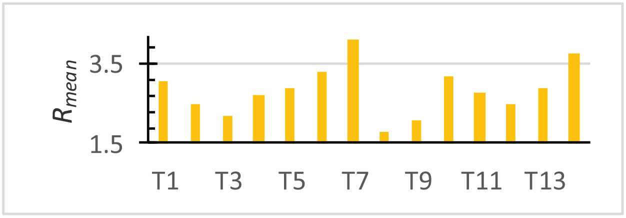
(a)4-core system



(b)8-core system



(c)16-core system



(d)32-core system

Fig. 5. Implementation cost analysis of locating victim caches in 4/8/16/32-core system

References

[1] M. Chaudhuri, "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches," IEEE/ACM. Symposium on Microarchitecture(MICRO), December 12-16, 2009, New York, NY, USA, pp. 401–412.

[2] L. R. Hsu, S. K. Reinhardt, R. Iyer, S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource," ACM. Conference on Parallel Architectures and Compilation Techniques(PACT), September 16-20, 2006, Seattle, Washington, USA, pp. 13–22.

[3] R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," ACM. Conference on Supercomputing(ICS), June 26-July 1, 2004, Saint-Malo, France, pp. 257–266.

[4] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. C. Steely, et al., “Adaptive insertion policies for managing shared caches,” ACM. Conference on Parallel Architectures and Compilation Techniques(PACT), October 25-29, 2008, Toronto, Canada, pp. 208–219.

[5] A. Jaleel, K. B. Theobald, S. C. Steely, J. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” ACM. Symposimum on Computer Architecture(ISCA), June 19-23, 2010, Saint-Malo, France, pp. 60–71.

[6] S. Kim, D. Chandra, Y. Solihin, “Fair cache sharing and partitioning in a chip multiprocessor architecture,” IEEE Computer Society. Conference on Parallel Architectures and Compilation Techniques(PACT), September 29-October 3, 2004, Antibes Juan-les-Pins, France, pp. 111–122.

[7] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, et al., “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” IEEE Computer Society. Conference on High-Performance Computer Architecture(HPCA), February 16-20, 2008, Salt Lake City, UT, USA, pp. 367–378.

[8] M. K. Qureshi, Y. N. Patt, “Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches,” IEEE Computer Society. Symposium on Microarchitecture(MICRO), December 9-13, 2006, Orlando, FL, USA, pp. 423–432.

[9] N. Rafique, W. T. Lim, M. Thottehodi, “Architectural support for operating system-driven CMP cache management,” ACM. Conference on Parallel Architectures and Compilation Techniques(PACT), September 16-20, 2006, Seattle, Washington, USA, pp. 2–12.

[10] D. Sanchez, C. Kozyrakis, “Vantage: scalable and efficient fine-grain cache partitioning,” ACM. Symposimum on Computer Architecture(ISCA), June 4-8, 2011, San Jose, CA, USA, pp. 57–68.

[11] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, et al., “Molecular caches: a caching structure for dynamic creation of applicationspecific heterogeneous cache regions,” IEEE Computer Society. Symposium on Microarchitecture(MICRO), December 9-13, 2006, Orlando, FL, USA, pp. 433–442.

[12] Y. Xie, G. H. Loh, “PIPP: promotion/insertion pseudo partitioning of multi-core shared caches,” ACM. Symposimum on Computer Architecture(ISCA), June 20-24, 2009, Austin, TX, USA, pp. 174–183.

[13] S. Srikantaiah, M. Kandemir, Q. Wang, “SHARP control: controlled shared cache management in chip multiprocessors,” ACM. Symposium on Microarchitecture(MICRO), December 12-16, 2009, New York, NY, USA, pp. 517–528.

[14] R. Manikantan, K. Rajan, R. Govindarajan, “Probabilistic shared cache management (PriSM),” ACM. Symposimum on Computer Architecture(ISCA), June 9-13, 2012, Portland, OR, USA, pp. 428-439.

[15] S. Eyerman, L. Eeckhout, “System-level performance metrics for multi-program workloads,” Micro, IEEE, 2008, vol. 28, no. 3, pp. 42 –53.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, J. Emer, “Adaptive insertion policies for high performance caching,” ACM. Symposimum on Computer Architecture(ISCA), June 9-13, 2007, San Diego, CA, USA, pp. 381–391.

[17] S. Eyerman, L. Eeckhout, T. Karkhanis, J. E. Smith, “A performance counter architecture for computing accurate CPI components,” ACM. Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS), October 21-25, 2006, San Jose, CA, USA, pp. 175–184.

[18] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, et al., “The M5 simulator: modeling networked systems,” Micro, IEEE, 2006, vol. 26, pp. 52–60.

[19] P. Kongetira, K. Aingaran, K. Olukotun, “Niagara: a 32-way multithreaded Sparc processor,” Micro, IEEE, 2005, vol. 25, no. 2, pp. 21-29.

[20] Z. Guz, I. Keidar, A. Kolodny, U. C. Weiser, “Utilizing shared data in chip multiprocessors with the Nahalal architecture,” the twentieth annual symposium on Parallelism in algorithms and architectures, 2008, pp. 1-10.

[21] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, et al., “A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS,” Solid-State Circuits Conference Digest of Technical Papers(ISSCC), 2010, pp. 108-109.

[22] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, et al., “On-Chip Interconnection Architecture of the Tile Processor,” Micro, IEEE, 2007, vol. 27, no. 5, pp. 15-31.