Adaptive Capacity Sharing through Probabilistic Controlled Placement

Xianju Yang
National University of Defense Technology, Changsha, China
Email: xianjuyang@nudt.edu.cn

Peixiang Yan
National University of Defense Technology, Changsha, China
Email: peixiangyan@gmail.com

Jiang Jiang
National University of Defense Technology, Changsha, China
Email: jiangjiang@nudt.edu.cn

Minxuan Zhang
National University of Defense Technology, Changsha, China
Email: mxzhang@nudt.edu.cn

***Abstract*—As capacity demands vary among simultaneously executed threads in chip multiprocessors, dynamically managing cache resources according to the run-time demands is effective to improve L2 cache performance. Differed from existing dynamic cache management schemes based on LRU replacement policy, we propose an adaptive capacity sharing mechanism based on a global reuse replacement policy. This mechanism adopts decoupled tag and data arrays, and partitions the data arrays into private and shared regions. Capacity sharing is accomplished by determining whether to place the incoming data into the private data region or into the shared data region, which is controlled by probabilities. Our mechanism includes: (1) A VMON monitor to predict run-time capacity demands. (2) A PCS algorithm to determine the probabilities. (3) A probabilistic controlled placement scheme to enforce capacity sharing. We evaluated our mechanism with a full system simulation of an 8-core CMP and used parallel programs from PARSEC benchmark suite. We found that with the same total L2 cache capacity, our mechanism exceeds the conventional private cache managed by LRU policy, the private cache without sharing managed by reuse replacement policy, and an existing adaptive sharing scheme based on LRU policy.**

***Index Terms*—Chip Multiprocessors, Capacity Sharing, Reuse Replacement.**

I. INTRODUCTION

Leveraging the ever broadening speed gap between processor and memory, performance of on-chip cache hierarchy is significant for further improvement of system performance. Development of CMP (Chip Multi-processors) imposes great stress on L2 cache hierarchy, as multiple threads are simultaneously executed. In CMP, L2 caches are usually organized in private or shared styles. With the same capacity, private cache organization has advantages of low access latency, flexible scalability and convenient performance isolation, while shared cache organization is benefit from large available capacity and simple consistency maintenance. As capacity demands vary greatly among programs and within programs, private cache organization suffers from limited available capacity, and shared cache organization confronts with malignant interference. Addressing these problems, capacity sharing [1]-[7] and partitioning [8]-[11] techniques become hot topics recently. We focus on capacity sharing based on private cache organization in this paper.

Existing capacity sharing mechanisms are based on LRU replacement policy. However, LRU policy is challenged by global replacement policies [12]-[14] in L2 caches. LRU policy picks out victims inside cache sets based on temporal locality and the locality of L2 accesses is weakened by the filter of L1 caches. Global replacement strategies are less restricted by the temporal locality within sets. Reuse replacement proposed in ref. [13] records the reuse counts of each data entry and sequentially search a data entry with zero reuse counts for eviction. As reuse replacement policy offers better resource utilization compared with LRU policy, we use reuse replacement to manage private L2 caches.

Based on reuse replacement policy, we propose a probabilistic controlled sharing mechanism, named PCS. PCS specifies part of data arrays as private and others as shared, and it employs probability to determine the frequency to place incoming data into the shared data region. We assign high probabilities to cores with stress capacity demands and dynamically adjust these probabilities according to the monitored memory demands. The main components of PCS mechanism are:

doi:10.4304/jcp.7.5.1236-1243

- A VMON monitor. Using a buffer to track recently evicted blocks of each core, run-time capacity demands are predicted by counting the number of cache access hits in the buffer.
- A PCS algorithm. According to the capacity demands predicted by VMON, PCS algorithm decides whether to promote or demote probabilities of each core.
- A probabilistic controlled placement scheme. Cores with higher probabilities will obtain more resources.

In our experiments, PARSEC benchmark suite [15] is used. PCS reduces the average L2 cache miss rate by 43.05% upon a conventional private LRU policy managed cache, by 8.70% upon a private reuse replacement policy managed cache, and by 16.42% upon an existing adaptive sharing mechanism.

The remainder of this paper is organized as follows. Section II illustrates PCS mechanism and section III introduces its implementation. We discuss our experimental methodology in section IV and present our results in section V. Section VI describes the related work and section VII concludes the paper.

II. Probabilistic Controlled Sharing Mechnism

A. Architectural Support

An 8-core CMP platform with private L2 caches managed by reuse replacement policy is depicted in Fig. 1. Dynamically allocating data entries to tag entries, tag and data entries are decoupled and linked by bi-directional pointers. The tag arrays are expanded as the total number of tag entries is larger than that of data entries. We partition the data arrays into private regions *P* and shard regions *S*, and then link all *S* to form a shared data region *sData*. Besides, PCS contains a VMON and a DAE (Data Access Engine). VMON monitors the run-time capacity demands and DAE decides which data region to access (*P* or *sData*).

Shown in Fig. 1, each tag entry includes *tag*, *status*, address of the corresponding data entry *d_ptr* and data region flag *s*. Excepting *data*, each data entry contains reuse count *reuse*, address of the tag entry *t_ptr* and valid bit *v*. For data entries in *S*, core identification *id* is appended. When one core occurs a L2 cache access, its tag array is searched and compared first. If the accessed tag entry is valid, the corresponding data entry is accessed using *d_ptr* and *s*. Otherwise, we search a data entry for eviction in *P* or *sData* according to *Prob*.

B. VMON Monitor

Memory demands of cores change not only among different programs, but also from one frame to another in the same program. Evaluation of run-time demands is of great importance to efficiently manage cache resources. Existing monitoring schemes [5][8][9] are accomplished by monitoring the tag array based on LRU replacement policy. These schemes are not appropriate for PCS based on reuse replacement policy mainly in two aspects. One is that existing schemes will get more complex and expensive when applying to PCS, as PCS varies ways in tag sets. The other one is that monitoring from the tag array cannot exactly reflect capacity demands, as PCS dynamically allocated data entries to tag entries. A new monitoring scheme named VMON is proposed in this paper. VMON uses a buffer to track recently evicted data blocks for each core. If one core has more hits in this buffer, it will benefit more from increasing capacity.

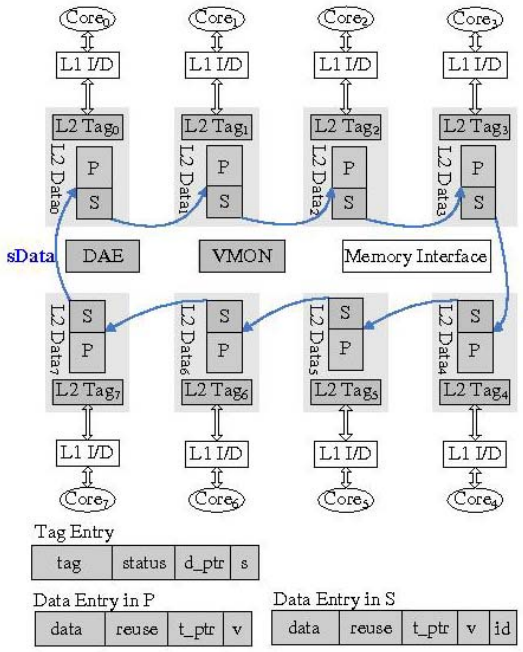


Figure 1. Architectural support for PCS mechanism.

In VMON, a buffer *VTag* is used to record tags of recently evicted blocks for each core. Performance gain for increasing capacity can be evaluated by the number of accesses that missed in L2 cache but hits in *VTag*. Suppose tags of recently evicted *G* blocks are recorded in *VTag*. When a L2 access misses in core *i* , check if it hits in $VTag_i$ and counts the number of hits in register $VTagHits_i$ during time interval *T*. Ideally, $VTagHits_i$ is the reduced L2 misses of core *i* when increasing its capacity by *G*. If $Access_i$ states the number of L2 accesses in core *i*, then the performance gain *MGi* can be calculated as shown in (1).

$$MG_i = \frac{VTagHits_i}{Access_i} \tag{1}$$

Cores with higher performance gain usually have higher capacity demands. However, VMON cannot evaluate performance lose for decreasing capacity. As PCS does not impose capacity decrease on cores with low capacity demands, but weaken their capability to use the shared resources, we simply treat cores with low performance gain as with low capacity demands, and believe it will have slightly influence on performance.

C. Probabilistic Controlled Sharing Algorithm

PCS implements dynamic capacity sharing among cores by controlling the frequency to place incoming data into the shared region. More placements in the shared data region, more capacity will be obtained. The

Algorithm Probabilistic Sharing

Input: $\{Prob_1, \cdots, Prob_K\}$, $\{VTagHits\}$, $\{Access\}$, $\{Prob_c\}$, Dec_Th, Inc_Th;
Output: $\{Prob_c\}$;
Variable: $\{MG\}$, AMG;
Description:
1. For i from 1 to N, calculate $MG_i = VTagHits_i / Access_i$
2. Calculate $AMG = \frac{1}{N}\sum_{i=1}^{N} MG_i$
3. For i from 0 to N-1, repeat:
   (a) If $MG_i > AMG * (1 + Inc\_Th)$ and $Prob_c^i \neq Prob_K$, then $Prob_c^i = Prob_{c+1}^i$;
   (b) If $MG_i < AMG * (1 - Dec\_Th)$ and $Prob_c^i \neq Prob_1$, then $Prob_c^i = Prob_{c-1}^i$.

Figure 3. Description of PCS algorithm.

placement frequency is determined by an evicting probability of shared resources, which is the probability to evict a data entry from the shared data region for the incoming data. For cores with higher capacity demands, we assign them higher evicting probabilities to obtain more shared resources. These probabilities are dynamically promoted or demoted according to the memory demands monitored by VMON.

Define *Prob* as the evicting probability of each core. Probability sharing algorithm is described in Fig.2, where {$Prob_1$, ⋯, $Prob_K$} are different levels of probabilities listed from low to high, {$Prob_c$} is the current evicting probability for each core, *Dec_Th* and *Inc_Th* are thresholds to downgrade and upgrade probabilities respectively. These thresholds are measured by the percentage deviating from average *MG*, since *MG* fluctuates heavenly with different programs. If *MG* of one core is beyond the *AMG* by *Inc_Th*, the evicting probability of that core is promoted. On the other hand, if *MG* of one core is below the *AMG* by *Dec_Th*, the evicting probability of that core is demoted.

III. Implementation

A. Framework

Shown in Fig. 3, PCS requires software and hardware cooperation, as PCS algorithm is implemented by OS, capacity demands monitor and capacity sharing enforcement is performed by hardware. Each time interval, VMON collects information and sends it to OS. Then, OS calculates new evicting probabilities $Prob_c$ for each core and writes them into DAE. Finally, DAE

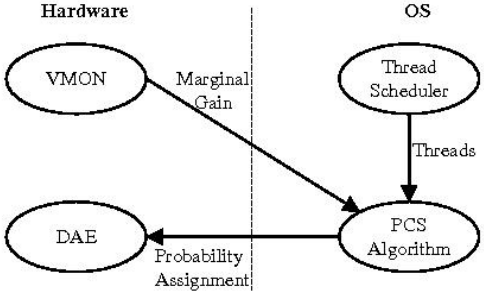


Figure 4. Framework of PCS mechanism.

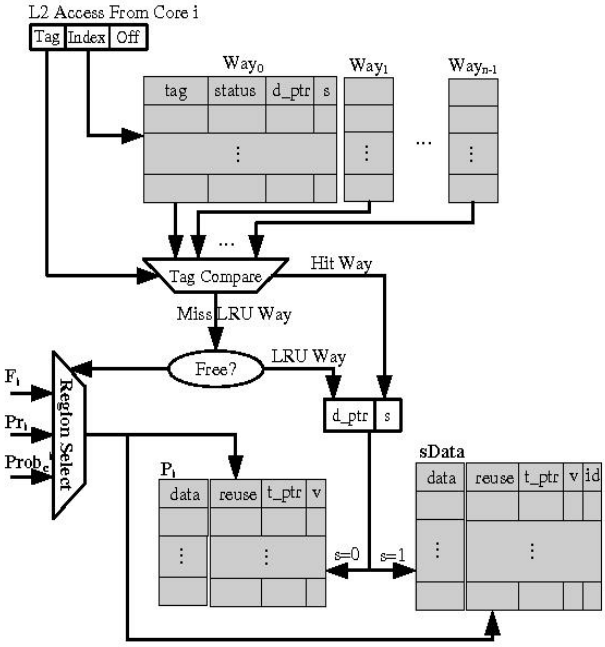


Figure 2. L2 cache access flow for core *i*.

accomplishes adaptive capacity sharing by probabilistically placing blocks into the shared data region.

B. Probability Controlled Placement

To enforce adaptive capacity sharing, DAE builds a probability generator PG for each core. When a L2 cache miss occurs and a data entry need to be evicted for core *i*, PG_i is launched to get a probability Pr_i. If Pr_i is not larger than $Prob_c^i$, then the incoming block will be placed into *sData*, otherwise placed into *P*. To utilize private resources, incoming blocks will not be placed into *sData* until *P* is full. Fig. 4 depicts the cache access flow in PCS mechanism.

However, it is impractical to implement completely probabilistic controlled placement, since PG for each core is expensive. Instead, we use private-to-shared ratios *PSR* to approximate probabilities. According to the *PSR* assigned to each core, we place incoming blocks into the private and shared data region in turn. Suppose $Prob_c^i = s_i/(p_i+s_i)$, then $PSR_i = s_i/p_i$. We place the first s_i incoming blocks into *sData*, the next p_i incoming blocks into *P*, and so on.

IV. Methodology

We use Virtutech Simics [16] to conduct our experiments, and simulate an 8-core CMP platform with sparc-v9 instruction set and solaris 10. A probability controlled PCS mechanism (PS-Reuse) and a PSR controlled PCS mechanism (PSR-Reuse) are simulated. To evaluate them, we also simulate a LRU managed private cache organization (P-LRU), a reuse replacement managed private cache organization (P-Reuse) and an existing LRU managed sharing mechanism (PS-LRU) proposed in ref. [5]. Table I shows the basic configurations of and different L2 cache mechanisms. Total miss rate and throughput is used to evaluate L2

TABLE II.
SIMULATION CONFIGURATION

Common	
System	1 chip, 8 cores
Core	Single-issue, in-order, no-branch-predictor
L1 I/D	32KB, 2-way, LRU, 64B/line, 1 cycle
Coherence	Snoop based MESI protocol
Main Memory	250 cycles
Different L2 Cache Mechanisms	
P-LRU	Traditional private cache, LRU, 126KB/core, 2048 entries/core, 2-way, 64B/line, 12/18 cycles
PS-LRU	Capacity sharing based on P-LRU, 1 way of each core for share, 12/18 cycles
P-Reuse	Private V-Way cache, reuse replacement, 4096 tag entries/core, 2048 data entries/core, 64B/line, 12/18 cycles
PS-Reuse	Our mechanism, sharing on P-Reuse, 64B/line, 12/18 cycles, probability controlled with step length 0.1 and default 0.5
PSR-Reuse	Oure mechanism, sharing on P-Reuse, 64B/line, 12/18 cycles, PSR controlled with probabilities {1/3, 1/2, 3/4} and default 1/2

cache and system performance. If MR_i and IPC_i state miss rate and IPC of core i respectively, these metrics can be calculated as follows:

$$MR_{sum} = \frac{1}{\#Cores} \sum_{i=0}^{\#Cores-1} MR_i \quad (2)$$

$$IPC_{sum} = \sum_{i=0}^{\#Cores-1} IPC_i \quad (3)$$

The average access latency of P-LRU is 12 cycles evaluated by Cacti [17] under 45nm technology. If a L2 cache access hits remote cores, we do not access memory but copy from the hit core. The average access latency for remote hits is 18 cycles. Assume sequential tag-data access is used for low power, the latencies of P-Reuse, PS-Reuse and PSR-Reuse are set as the same as P-LRU.

In our experiments, the PARSEC 2.1 benchmark suite is used. Table II describes the function of these programs except *dedup*, which is excluded as it cannot run normally due to compatibility. All programs use *simlarge* input sets and have 8 threads running concurrently. Each core runs the first 200 million instructions to skip initial stage and warm up cache hierarchies, and then runs the next 200 million instructions for evaluation. In all, about 3.2 billion instructions are executed for 8 cores.

TABLE I.
PARSEC PROGRAMS

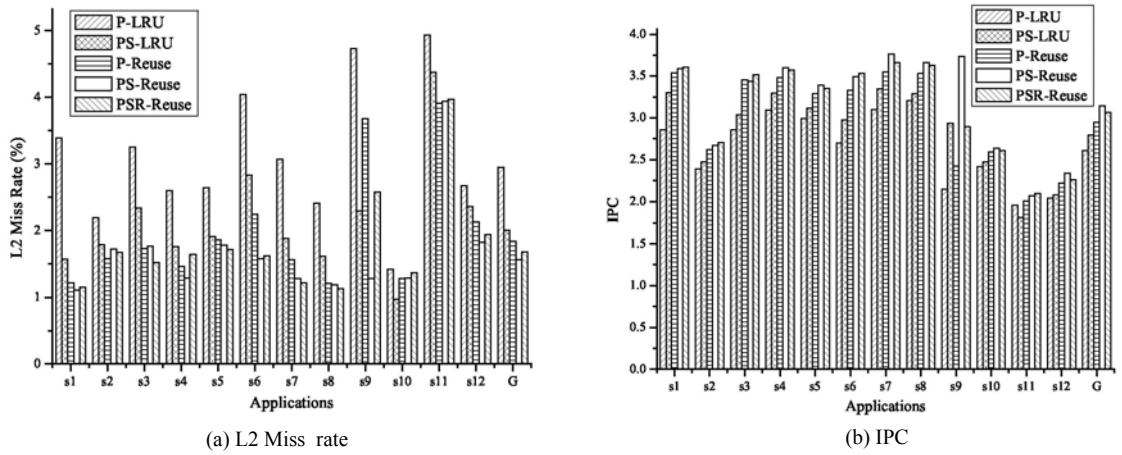
Label	Program	Descriptions
s1	blackscholes	Calculate portfolio price using Black-Scholes PDE
s2	bodytrack	Computer vision, tracks 3D pose of human body
s3	canneal	Synthetic chip design, routing
s4	facesim	Physics simulation, models a human face
s5	ferret	Pipelined audio, image and video searches
s6	fluidanimate	Physics simulation, animation of fluids
s7	freqmine	Data mining application
s8	raytrace	Computer animation application
s9	streamcluster	Kernel to solve the online clustering problem
s10	swaptions	Computers portfolio prices using Monte-Carlo simulation
s11	vips	Image processing, image transformations
s12	x264	H.264 video encoder

V. RESULT

A. Performance Analysis

The L2 cache miss rate and total IPC of PS-Reuse and PSR-Reuse are compared with that of P-LRU, PS-LRU and P-Reuse. In PS-Reuse and PSR-Reuse, the private data size is 1024 entries, the upgrade and downgrade thresholds are both 50%, the time interval is 10 million instructions, and VMON has 256 entries for each core by default.

Fig. 5(a) compares the L2 cache miss rates of P-LRU, PS-LRU, P-Reuse, PS-Reuse and PSR-Reuse. Label *s1* to



(a) L2 Miss rate (b) IPC

Figure 5. Performance comparisons of different L2 cache mechanisms.

s12 represents applications and label *G* is the geometric average value. The average miss rates of these five mechanisms are 2.95%, 2.01%, 1.84%, 1.57% and 1.68% respectively. Conclusions of these simulation results are as following.

- Expanding capacity sharing on private organization can improve cache performance. The average L2 miss rate of PS-Reuse is reduced by 46.78% compared with P-LRU and by 14.67% compared with P-Reuse. The average L2 miss rate of PSR-Reuse is reduced by 43.05% compared with P-LRU and by 8.70% compared with P-Reuse.
- Mechanisms with reuse replacement policy gain better performance than mechanisms with LRU replacement policy. Compared with PS-LRU, PS-Reuse reduces the average L2 miss rate by 21.89%, and PSR-Reuse reduces the average L2 miss rate by 16.42%.
- PS-Reuse outperforms PSR-Reuse in L2 cache miss rate, as PS-Reuse adopts a more flexible random placement scheme.

The simulation results also demonstrate that reduction in L2 cache miss rates can also improve system performance. IPCs of P-LRU, PS-LRU, P-Reuse, PS-Reuse and PSR-Reuse are depicted in Fig. 5(b). Their average IPC are 2.609, 2.84, 2.946, 3.141 and 3.066 respectively. The average IPC of PS-Reuse is improved by 20.39% upon P-LRU, 10.60% upon PS-LRU, and 6.62% upon P-Reuse. The average IPC of PSR-Reuse is improved by 17.52% upon P-LRU, 7.77% upon PS-LRU, and 4.07% upon P-Reuse.

B. Parameters Sensitivity

Probability levels, upgrade and downgrade thresholds, private data size, time intervals and VMON size are five parameters in PCS. Based on the default settings of PSR-Reuse, we change one parameter at a time to observe its effects on L2 miss rate.

1) Probability Levels: Standing for the opportunity to place incoming data into the shared data region, the probability of each core changes from 0 to 1 ideally and must be set according to the capacity demands. If probabilities are too high to beyond demands, blocks in the private data region will become dead blocks as incoming blocks are mostly placed into the shared region and private blocks are rarely evicted. Otherwise, if probabilities are too low to below demands, cores with stress capacity demands are unable to obtain enough resources. In our experiments, we randomly select probabilities from [1/3, 2/5, 1/2, 3/5, 2/3, 3/4] to form eight different cases and observed their influences on L2 miss rates. Probabilities are all initialized to 1/2 with equal opportunities to use private and shared resources. Shown in Fig. 6, it is hard to decide the optimize setting of probability levels as memory demands vary greatly. Generally, applications with large demand differences among cores are benefit from coarse probability levels, and applications with small demand differences prefer fine probability levels.

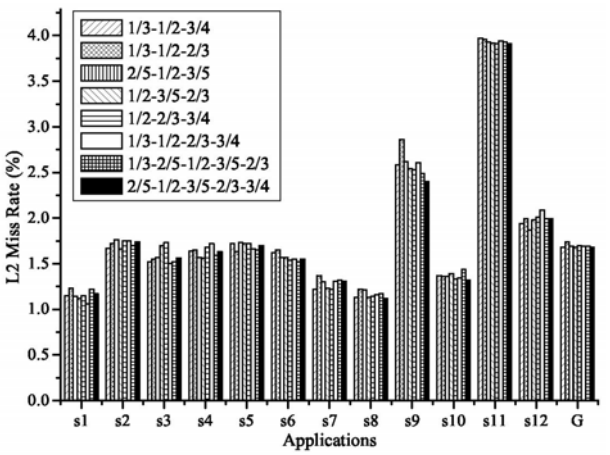


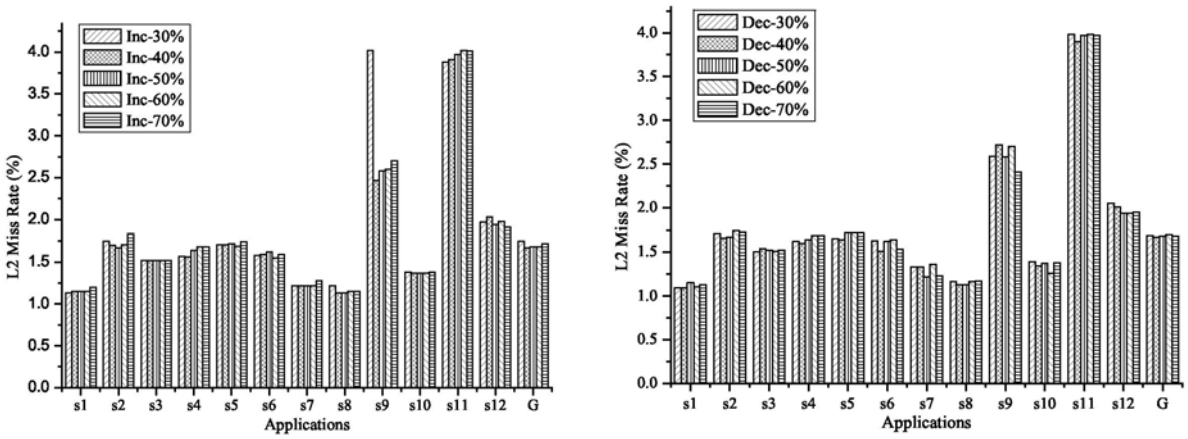
Figure 6. Effects of different levels of probability on L2 miss rate.

2) Upgrade and Downgrade Thresholds: Upgrade and downgrade thresholds influence the adjustments of probabilities. If the upgrade threshold is too low or the downgrade threshold is too high, probabilities tend to be high which will cause frequent utilization of the shared resources and induce aggravation of interferences among cores. Otherwise, if the upgrade threshold is too high or the downgrade threshold is too low, probabilities tend to be low which will cause infrequent utilization of the shared resources and cause difficulties in obtaining enough capacity. We observe the effects of upgrade and downgrade thresholds by changing them from 30% to 70%. It can be seen from Fig. 7(a) and Fig. 7(b) that both upgrade and downgrade thresholds should be moderate. In our experiments, upgrade threshold of 40% gain better average performance. Downgrade threshold does not affect much relative to upgrade threshold.

3) Private Data Size: Private data size is related to the amounts of resources for sharing. Maintaining the total size of L2 cache, the smaller the private data region, the

TABLE III.
EFFECTS OF ENTRIES IN VMON ON L2 MISS RATE (%)

No.(entries)	*s1*	*s2*	*s3*	*s4*	*s5*	*s6*	*s7*	*s8*	s9	*s10*	*s11*	*s12*	*G*
32	1.09	1.74	1.90	1.72	1.70	1.52	1.22	1.18	2.58	1.37	3.93	2.02	1.83
64	1.14	1.67	1.50	1.73	1.63	1.53	1.22	1.19	2.39	1.36	3.97	1.95	1.77
128	1.15	1.66	1.53	1.59	1.71	1.51	1.31	1.21	2.61	1.38	3.96	2.00	1.80
256	1.15	1.67	1.52	1.64	1.72	1.62	1.29	1.13	2.58	1.37	3.97	1.94	1.80



(a) Upgrade thresholds (b) Downgrade thresholds

Figure 7. Effects of thresholds on L2 miss rate.

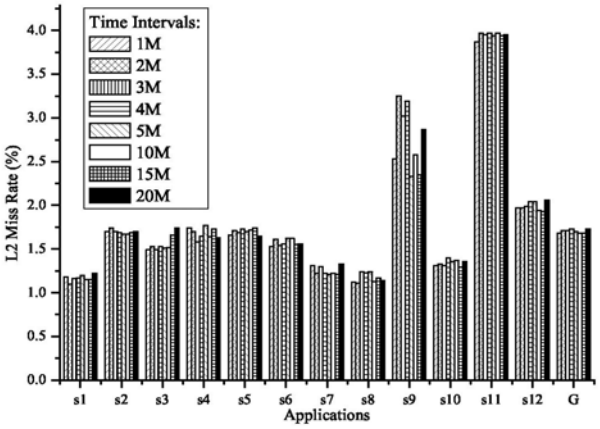


Figure 8. Effects of time intervals on L2 miss rate.

larger the shared data region. Proper size of private region relies on the differences of capacity demands among cores. Applications with large demand differences are benefit from large shared region, and applications with little demand differences will prefer large private region. We compare private data size settings of 512, 1024 and 1536 entries in Fig. 8. *s5*, *s8* and *s11* get better performance with 512 private entries, s3, s6 and s7 get better performance with 1024 private entries, and other applications prefer 1536 private entries.

4) Time Intervals: Time intervals should set moderately as large time intervals will affect the adaptability to variable memory demands, and small time intervals do not have sufficient time to collect reliable statistics. We compare L2 miss rates under different time intervals of 1, 2, 3, 4, 5, 10, 15 and 20 million instructions, shown in Fig. 9. The optimize time intervals for each program is different, and time intervals of 10 million instructions has slightly better average performance.

5) VMON Size: Recording recently evicted blocks to predict capacity demands, VMON should have enough space to hold these blocks. However, VMON size is not bigger better for two reasons. When VMON is big enough, more hits in VMON cannot gain better performance for increasing capacity, as allocation of data entries to a tag set is limited by the tag size. The other one is that bigger VMON will induce greater area overhead. Changing the VMON size among 32, 64, 128 and 256 entries for each core, table III shows their effects on L2 miss rates. The setting with 64 entries achieves better performance than other settings in our experiments. Although the optimize settings of VMON size vary with applications, their effects on L2 miss rate are small.

C. Storage Overhead

Expanding sharing mechanism on P-Reuse, PS-Reuse attaches a *s* flag in each tag entry, an *id* field in each shared data entry, and a *Vtag* buffer for each core. Suppose the memory address is 64b, the line size is 64B and the *Vtag* size is 64 entries. The storage cost of P-LRU, P-Reuse and PS-Reuse are about 1126KB, 1306KB and 1324KB respectively. That is, P-Reuse increases the storage overhead by 15.99% compared with P-LRU, and PS-Reuse further increases the storage overhead by 1.38% upon P-Reuse.

VI. Related Work

A. Capacity Sharing Techniques

There are mainly two basic ways to expand capacity sharing based on private cache organization. One is implemented by migrating blocks among cores [1]-[4] and the other one is implemented by specifying shared regions [5]-[7].

Migrating blocs among cores, Chishti et al. [1] proposes capacity stealing technique for private blocks. Cores with large capacity demands can place least recently used private blocks into their neighboring cores. Chang et al. [2] proposes CC (Cooperative Caching) mechanism. CC employs cache-to-cache transfers of clean data, replication-aware data replacement, and global replacement of inactive data to form an aggregate "shared" cache. DSR (Dynamic Spill-Receive) mechanism [4] exploits application-level capacity demands and SNUG (Set-level Non-Uniformity identifier and Grouper) [3] exploits set-level capacity demands to allow blocks migrate from cores with higher demands to cores with lower demands.

Partitioning local caches into private and shared region, Dybdahl et al. [5] uses sharing engine to control the utilization of shared resources and dynamically adjust the private capacity, Zhao et al. [7] attempts to provide more hits into local slice for workloads with no sharing and supply more sharing resources for workloads with sufficient sharing.

This paper is the first work to exploit capacity sharing based on reuse replacement policy. Mechanism propose in ref. [5] is compared with our work in section V-A.

B. Global Replacement Policies

As LRU replacement policy is limited to picking out victims inside sets, many researches are dedicated to exploiting global replacement policies to manage L2 cache resources.

Generational replacement policy proposed in Ref. [12] divides cache blocks into several priority groups. Frequently accessed blocks are put into groups with higher priority and infrequently accessed blocks are put into groups with lower priority. Generational replacement policy picks out blocks from the lowest priority groups.

Based on reuse counts, Qureshi et al. [13] proposes reuse replacement policy. Reuse replacement policy attaches a reuse counter to each cache block and maintains a global searching pointer. The reuse counter is initialized to 0, plused 1 for a cache access hit to that block, and minused 1 for the searching pointer passes through that block. When evicting a data entry, reuse replacement policy circularly searches from the next position of the global pointer until a block with zero reuse counts is found.

Rajan et al. [14] separates the L2 cache into SC (Shepherd Cache) and MC (Main Cache) to emulate optimal replacement policy. SC adopts FIFO (First In First Out) replacement policy and guide replacement operations in MC.

Among them, generational replacement and reuse replacement are more flexible than SC, as tag and data entries are decoupled. Considering that generational replacement is software implemented and confronts with long latency, we use reuse replacement policy to manage L2 cache resources.

VII. CONCLUSION

Using reuse replacement strategy to manage private L2 cache resources, we propose a probabilistic sharing mechanism named PCS. PCS separates tag and data arrays, and specifies a data region as shared region. All cores can use resources in the shared data region. PCS uses probabilities to control the utilization of shared data resources. Cores with stress demands are assigned higher probabilities to obtain more shared resources. According to the run-time capacity demands monitored by a VMON scheme, these probabilities are adjusted every time interval. As probability generator is expensive to implement, we uses alternately placing into private and shared region according to a specified ratio to approximate probabilistic controlled placement. Simulation results with programs from PARSEC benchmark suit show that, our mechanism reduces the average L2 miss rate by 43.05% compared with a conventional LRU managed private cache organization, by 8.70% compared with a reuse replacement managed private cache organization, by 16.42% compared with an existing LRU based capacity sharing mechanism. System performance is also effectively improved by our mechanism. We plan to extend this work by balancing tag set utilization and by developing tag sharing among cores. Work is also underway in varying private data size with applications.

ACKNOWLEDGMENT

This work is supported by the National Nature Science Foundation of China under NSFC No.60970036, No. 60873016 and No.61103011, 863 Project of China under contract 2009AA01Z124.

REFERENCES

[1] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA '05*: *Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 357--368. doi:10.1109/ISCA.2005.39.

[2] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 264--276. doi:10.1109/ISCA.2006.17.

[3] D. Zhan, H. Jiang, and S. Seth, "Exploiting set-level non-uniformity of capacity demand to enhance cmp cooperative caching," in *IPDPS'10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010, pp. 1--10. doi:10.1109/IPDPS.2010.5470441.

[4] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in cmps," in *International Symposium on High-Performance Computer Architecture*, 2009, pp. 45--54. doi:10.1109/HPCA.2009.4798236.

[5] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2--12. doi:10.1109/HPCA.2007.346180.

[6] T. Y. Yeh and G. Reinman, "Fast and fair: data-stream quality of service," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2005, pp. 237--248. doi:10.1145/1086297.1086328.

[7] L. Zhao, R. Iyer, M. Upton, and D. Newell, "Towards hybrid last level caches for chip-multiprocessors," *SIGARCH Comput. Archit. News*, vol. 36, pp. 56--63, May 2008. doi:10.1145/1399972.1399982.

[8] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, pp. 7--26, April 2004. doi:10.1023/B:SUPE.0000014800.27383.8f.

[9] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA:

IEEE Computer Society, 2006, pp. 423--432. doi:10.1109/MICRO.2006.49.
[10] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111--122. doi:10.1109/PACT.2004.15.
[11] R. Iyer, "Cqos: a framework for enabling qos in shared caches of CMP platforms," in *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2004, pp. 257--266. doi:10.1145/1006209.1006246.
[12] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 107--116, May 2000. doi:10.1145/342001.339660.
[13] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 544--555. doi:10.1109/ISCA.2005.52.
[14] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 445--454. doi:10.1109/MICRO.2007.14.
[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* New York, NY, USA: ACM, 2008, pp. 72--81. doi:10.1145/1454115.1454128.
[16] P. S. Magnusson, M. Christensson, J. E. and et al., "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50--58, 2002. doi:10.1109/2.982916.
[17] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches," University of Utah and Hewlett Packard Laboratories, Tech. Rep., 2009.

Xianju Yang was born in Hunan province of China in 1980. He received the M.S. degree from National University of Defense Technology in 2005. Now he is a Ph.D. candidate. His research interests include micro-processor design and microelectronics.

Peixiang Yan was born in Hunan province of China in 1981. Now she is a Ph.D. candidate. Her research interests include microprocessor design and micro-electronics.

Jiang Jiang was born in Yunnan province of China. Now he is an associate professor in National University of Defense Technology. His research interests include computer architecture, microprocessor design, ASIC design and FPGA acceleration.

Minxuan Zhang was born in Hunan province of China in 1954. Now he is a professor in National University of Defense Technology. His research interests include computer architecture, microprocessor design, low power and ASIC design.