EFFICIENT CACHE PARTITIONING TECHNIQUE FOR CHIP MULTIPROCESSORS

Prof H. R. Deshmukh*

Dr. G. R. Bamnote**

ABSTRACT

Chip multiprocessors (CMPs) have been widely adopted and commercially available as the building blocks for future computer systems. It contains multiple cores, which enables to concurrently execute multiple applications (or threads) on a single chip. As the number of cores on a chip increases, the pressure on the memory system to sustain the memory requirements of all the concurrently executing applications (or threads) increases. An important question in CMP design is how to use the limited size L2 cache on chip to achieve the best possible system throughput for a wide range of applications. Keys to obtaining high performance from multicore architectures is to provide fast data accesses (reduce latency) for on-chip computation resources and manage the largest level on-chip L2 cache efficiently so that off-chip accesses are reduced.We propose efficient cache partitioning (ECP) technique in which the amount ofL2 cache space that can be shared among the cores is controlled dynamically. Efficient cache partitioning (ECP) technique estimates, continuously, the effect of increasing/decreasing the shared partition size on the overall Performance. We show that our partitioning technique performs better than traditional techniques like LRU partitioning and Half-and-Half partitioning under Efficient Replacement Policy.

*Associate professor, B.N.C.O.E. Pusad(M.S.).

**Professor & Head, PRMIT&R, Badnera(M.S.).

I. INTRODUCTION

Main challenges for next generation microprocessors are the slow main memory and the limited off-chip bandwidth. Efficient management of the last level L2 on-chip cache is therefore important in order to accommodate larger number of cores in future multi-core architectures.

Traditional design for on-chip cache uses the LRU partitioning technique as well as Half-and-Half partitioning technique. LRU partitioning technique implicitly partitions a shared cache among the competing applications on a demand1 basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, the benefit that an application gets from cache resources may not directly correlate with its demand for cache resource. For example, a streaming application can access a large number of unique cache blocks but these blocks are unlikely to be reused again if the working set of the application is greater than the cache size. Although such an application has a high demand, devoting a large amount of cache will not improve its performance. Thus, it makes sense to partition the cache based on how much the application is likely to benefit from the cache rather than the application's demand for the cache. However the Half-and-Half scheme statically partitions the cache equally among the two competing applications. The disadvantage of the Half-and-Half scheme over LRU is that it cannot change the partition in response to the varying demands of competing applications

In efficient cache partitioning technique we divide L2 cache into shared as well as private partition for each core that aims at combining the best of the two extreme organizations by combining the low latency of small (private) caches with the capacity of a larger (shared) cache. In our partitioning technique, hits to the private partitions are fast, while hits to shared partitions are slower. The size of the private and shared partition is dynamically controlled and balanced against the other cores. Cores that can best utilize the cache get more private cache space, but the private cache space is never larger than the local last level cache. The cache usage in the shared partition is controlled as well. The size of the private partitions for each core is dynamically adjusted to minimize the total number of cache misses

We compare the performance of the new scheme with the performance of traditional partitioning techniques like LRU partitioning as well as Half-and-Half partitioning technique. Additionally, we also compare performance of three techniques under traditional least recently used replacement policy and efficient replacement policy. We show that new scheme performs better than traditional partitioning schemes, as well as all three techniques perform better under efficient replacement policy rather than traditional replacement policy like LRU. The rest of the paper is organized as follows. Section 2 defines new scheme. Section 3 presents the evaluation methodology. Section 4 presents results. Related work is discussed in section 5 and we conclude in section 6.

II. THE EFFICIENT CACHE PARTITIONING TECHNIQUE (ECP)

An architectural structure for a two-core system with the proposed system is shown in Figure 1. This structure is a abstract view rather than implying a physical layout. Each core has two levels of cache, where the L2 cache is the last-level cache. The partitioning technique implements sharing of the last-level cache between the cores. Hits in the core private L2 cache partition are faster than the hits in the neighboring last-level shared cache partitions. The usage of L2 cache is controlled in two ways one is a part of the L2 cache is private and inaccessible by the other cores, and second is the number of cache blocks in the private cache partition and the number of cache blocks in the shared partition of the cache are controlled on a per-core basis in order to minimize the total number of cache misses. L2 cache for each core is divided in to two partitions a private partition for its own cache blocks and a shared cache partition intended for all the cores. The most recently used cache blocks for each core are stored in the private partition.

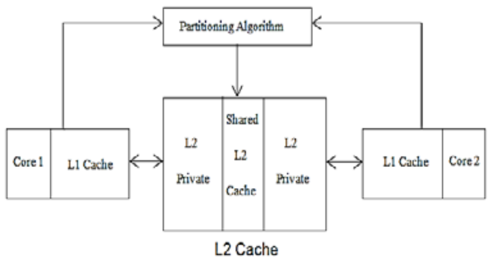

Figure 1: Framework for Efficient Cache Partitioning Technique

A. L2 Cache partition Management

Each L2 cache is divided into a private and a shared partition as shown in Figure 1. Initially 75% of L2 cache partitions are private while 25% partition of L2 cache acts as a shared partition for all cores. The private partition is not shared and is managed by a least recently used(LRU) replacement policy or efficient replacement policy for comparison of our scheme under both the replacement policy. To find a victim using LRU or ERP the private partition is only considered and blocks belonging to the shared partition are not involved. While eviction of blocks from shared cache LRU or ERP replacement policy will be used. If the cache hit in private portion of L2 cache, the block that is hit is moved to the private of L1 cache, and no access to the shared partition of the L2 cache is required. If a miss occurred in the private partition of the L2 cache, all the L2shared caches partitions are checked in parallel since the cache block can be in any of these caches. The cache block with the hit in shared cache is moved to the private L2 cache of requesting core. If the private L2 cache is full the blocks according to LRU or ERP cache block in the private cache replaces the block with a hit in the shared cache, and the replaced block is set as MRU in the shared cache. If block neither available in private nor in shared of L2,cache miss occur. On occurrence of cache miss the block is requested from main memory and inserted into the private cache.

B. L2 Cache Partitioning Technique

For efficient utilization of L2 cache, cache is divided in to shared and private partition. Size of both partitions are dynamically increased or decreased, based on how much application benefit from cache rather than on its demand for cache. We propose efficient cache partitioning scheme for dynamic partitioning of L2 cache. This technique monitors each application at runtime and collected run time information. This collected run time information for each core is a input to partitioning algorithm to decide the amount of cache resources allocated to each core. For collection of run time information we used Replace_Block_Tag, Replace_Block_Tag_Counter, and Private_Cache_Hit_Counter for each core. When cache block is evicted from shared L2 Cache, only the tag of block is stored in appropriate Replace_Block_Tag not a block data. Replace_Block_Tag is equal to number of blocks initially in private cache. On a cache miss tag of miss is compared to appropriate Replace_Block_Tag. If there is match, Replace_Block_Tag_Counter is incremented. Appropriate Private_Cache_Hit_Counter is increase if hit is occurred in appropriate private L2 cache for requesting core. While running process if private and shared cache is full and tag of block neither present in private nor in shared but present in Replace_Block_Tag, then Private_Cache_Hit_Counter and Replace_Block_Tag_Counter is checked. If Private_Cache_Hit_Counter is less than Replace_Block_Tag_Counter then the size of private L2 cache is increased by one block for requesting core and size of shared cache block is decreased by one block. This condition represent that core don’t have sufficient private memory to run process. If we increase the size of Private there is chance of increasing the hit for requesting core. If Private_Cache _Hit_Counter is greater than Replace_Block_Tag_Counter then the block of shared will increase by one block and private will be reduced by one block for requesting core. This condition represent that core have sufficient private memory to run the process. If we reduce the block of private, hit rate of requesting core will not be affected and other core will get spaced if required to store his blocks. All the appropriate counters will be reset after every increased or decreased in blocks of private or shared cache in requesting core. When content of requested block is transfer from Replace_Block_Tag to private, while transfer copy of requested block is removed from Replace_Block_Tag. When block is transfer from shared to private content of shared transfer to Replace_Block_Tag as per LRU or ERP replacement policy. When block is transfer from private to shared content of transfer block stored to Replace_Block_Tag. In this way dynamic partitioning will be done in efficient cache partitioning technique.

C. Discussion

As we have divide L2 cache in to private and shared partitions, we got high performance from multicore by reducing latency due to division of L2 cache into private cache as private memory required low latency to get a block into the cache. Efficient management of L2 cache reduces off chip accesses which increase the bandwidth and speedup the application. Our scheme solved the problem of uncontrolled contention on shared cache due to division of L2 cache on per core basis. One implication of using efficient partitioning is that some applications do not get the cache space they require because some other applications can utilize the cache more efficiently in the sense that the total miss rate will be lower. This means that an application that frequently accesses the last-level cache and which benefits from a large cache space is likely to get more cache capacity. An application that infrequently accesses the last-level cache, but that would also experience a lower miss rate from a larger cache space will less likely get a larger cache space since the number of misses removed by increasing the cache space for that application is lower. Consequently, applications that access the cache rarely or that do not benefit from a larger cache space will receive modest cache capacity if other more demanding cores (on the same chip) benefit from a large cache space.

D. Replacement policies

This partitioning technique use two replacement policy one is least recently used (LRU) and other is efficient replacement policy (ERP). Performance of partitioning scheme is observed under both the replacement policies and it is find out that for most of the cases efficient replacement policy work well in efficient cache partitioning technique. On cache misses, data loaded from main memory is always allocated in the private partition of the cache as most recently used. This normally requires that one block is evicted from the private partition of the cache. The evicted block is allocated in the shared cache partition. Eviction of blocks from private as well as shared done through following two policies.

a. LRU replacement policy

The LRU replacement policy replaces the least recently used pages in the cache. It only maintains the recency of pages. LRU policy uses the recent past as an approximation of the near future, and therefore replaces the line that has not been used for the longest period of time. The LRU performs well in some applications, but it isn't able to cope with access patterns such as file scanning, regular accessed over more pages than the cache size, and accessed on pages with distinct frequencies. LRU has a high overhead cost of moving cache blocks into the most recently used position each time a cache block is accessed. LRU does not use 'frequency' information of memory accesses and LRU is prone to cache pollution when a sequence of single-use memory accesses that are larger than the cache size is fetched from memory. This policy fails for sequential access.

b. Efficient Replacement policy

ERP tries to replace the page which are not referenced more often. To implement the ERP cache replacement policy, we created a pointer called *Replace_Ptr*and an array of *Hit_Bit called* reference bits for each cache block in a cache set. The ERP policy uses a circular buffer with the *Replace_Ptr* pointing to the cache block that is to be replaced when a cache miss occurs. The use of the circular queue avoids the movement of cache blocks from the head of the queue to the tail of the queue; instead it replaces the block by advancing the *Replace_Ptr*to point to the next cache block in the circular queue. Replace_Ptr will only be advance by resetting hit bit when hit bit of page that Replace_Ptr is pointing is 1. During a cache hit, the ERP policy will set the *Hit_Bit*of the accessed cache block to 1 to indicate that the cache block has been hit. When a cache miss occurs, *Replace_Ptr will* not to advance to the next cache block whenever the *Hit_Bit*of the cache block pointed by the *Replace_Ptr*is equal to '0' and new cache block will be placed at Replace_Ptr position. Initially reference bit is 0, policy sets it to 1 as soon as the corresponding cache block is referenced. Reference bit = 0 means that the cache block has not been referenced and hence, it can be replaced. Reference bit = 1 means the corresponding cache block has been referenced and hence, is likely to be used soon therefore, it is not replace. The main purpose in the development of the ERP is to create a cache replacement policy that has lower maintenance cost compared toLRU replacement policies.

E. Efficient cache Partitioning Algorithm

```
Step 1:START
Step 2: INPUT Cache size (N)
        INPUT Number of Cores (NC)
```

```
Step 3: Set Private P=(N*0.75)/CN
        Set Shared S=(N*0.25)/CN
Step 4: CPU generate addresses X for each Core
Step 5: If block X is in Private of L2
        {
                PRIVATE_CACHE _HIT_COUNTER ++;
                Process block X
        }
Step 6: If block X in Shared of L2
        {
                Transfer block X to Private of L2
                Process block X
        }
Step 7: If block X is in Neighboring Shared of L2
        {
                Transfer block X to Private of L2
                Process block X
        }
Step 8: If block X is in Replace_Block_Tag
        {
        REPLACE_BLOCK_TAG_COUNTER ++
        If
        PRIVATE_CACHE_HIT_COUNTER<REPLACE_BLOCK_TAG_COUNTER
                {
                Increment One Block of Private L2
                Decrement One Block of Shared L2
                Fetch block X from Main Memory(MM)
                Place block X at Private L2
                Process block X
                Reset PRIVATE_CACHE_HIT_COUNTER
                and
                REPLACE_BLOCK_TAG_COUNTER
                }
                Else
```

```
          {
          Increment One Block of Shared L2
          Decrement One Block of Private L2
          Fetch block X from Main Memory(MM)
          Place block X at Private L2
          Process block X
          Reset
          PRIVATE_CACHE _HIT_COUNTER
          and
          REPLACE_BLOCK_TAG_COUNTER
          }
     }
Step 9: If block X is not in Replace_Block_Tag
     {
          If Free Space in Private L2
          {
          Fetch block X from Main Memory(MM)
          Place block X in Private of L2
          Process block X
          }
          If Free Space in Shred L2
          {
          Apply  Replacement Policy and Transfer Evicted block from Private to Shared
          Fetch block X from MM
          Place block X at Private L2
          Process block X
          }
          If Free Space is not in Shared L2
          {
          Apply  Replacement Policy and Transfer Evicted block from Shared to
          Replace_Block_Tag
          Apply  Replacement Policy and Transfer Evicted block from Private to Shared
          Fetch block X from MM
          Place block X at Private L2
```

```
        Process block X
        }
    }
Step 10 :STOP
```

III. CONVENTIONAL PARTITIONING ALGORITHMS

To check the performance of new cache partitioning scheme, we have used conventional LRU cache partitioning and Half-and-Half cache partitioning Technique. The LRU policy implicitly partitions a shared cache among the competing applications on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, the benefit that an application gets from cache resources may not directly correlate with its demand for cache resource. For example, a streaming application can access a large number of unique cache blocks but these blocks are unlikely to be reused again if the working set of the application is greater than the cache size. Although such an application has a high demand, devoting a large amount of cache will not improve its performance. Thus, it makes sense to partition the cache based on how much the application is likely to benefit from the cache rather than the application's demand for the cache. Demand is determined by the number of unique cache blocks accessed in a given interval. Consider two applications A and B containing N blocks. Then with LRU replacement, the number of cache blocks that each application receives is decided by the number of unique block accessed by each application in the last N unique accesses to the cache. If UA is the number of unique blocks accessed by application A in the last N unique accesses to the cache, then application A will receive UA cache blocks out of the N blocks in the cache.

The Half-and-Half scheme statically partitions the cache equally among the two competing applications. The Half-and-Half scheme is cannot change the partition in response to the varying demands of competing applications. However, it has the advantage of performance isolation, which means that the performance of an application does not degrade substantially when it executes concurrently with a badly behaving application.

IV. RESULTS AND ANALYSIS

→ POLICIES / WORKLOADS ↓	% Hit					
	LRU Replacement Policy			Efficient Replacement Policy		
	Half-and-Half	LRU	ECP	Half-and-Half	LRU	ECP
W1	25	30	35	30	30	40
W 2	20	20	25	25	25	30
W 3	25	35	35	30	30	40
W 4	40	40	40	40	40	45
W 5	20	25	25	25	25	25
W 6	45	50	55	50	50	55
W 7	25	25	35	30	30	40
W 8	30	35	40	30	40	45
W 9	50	55	55	55	55	60
W10	40	35	45	35	40	45

Table 1: % Hit for different partitioning techniques under different replacement policies.

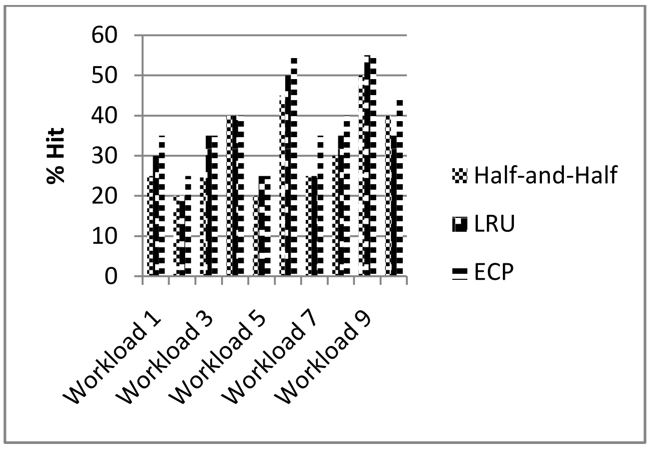

Figure 2: % Hit for Half-and-Half, LRU, and ECP Cache partitioning under LRU replacement policy

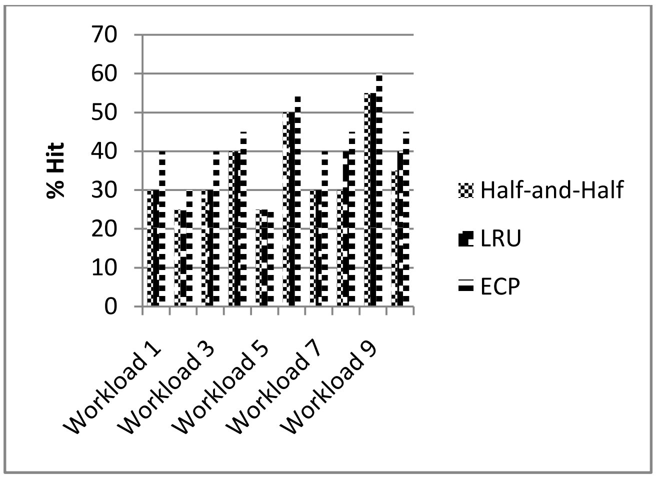

Figure 3: % Hit for Half-and-Half, LRU, and ECP Cache partitioning under Efficient Replacement Policy.

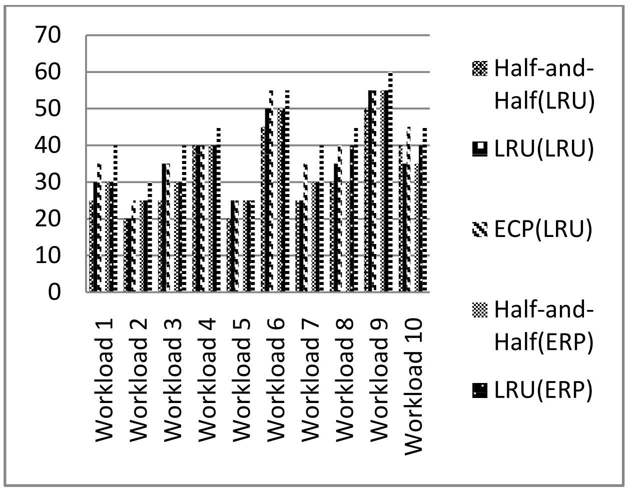

Figure 4: % Hit for Half-and-Half, LRU, ECP under LRU and ERP Replacement Policy

We compare the performance of new cache partitioning technique i.e. Efficient Cache Partitioning (ECP) to two conventional partitioning scheme: Half-and-Half & LRU partitioning under LRU replacement policy & Efficient Replacement policy. Figure 2 shows percentage hit for Half-and-Half, LRU partitioning &Efficient Cache Partitioning Technique under LRU replacement policy & it is observed that LRU partitioning perform better than Half-and-Half for some work load & perform equal for few workloads, but for most of the workload our new technique ECP outperform the best performing scheme out of other two conventional scheme. On average the ECP improves performance by 5.50% on conventional scheme under LRU replacement policy.

Figure 3 shows percentage hit for Half-and-Half, LRU partitioning & Efficient Cache Partitioning Technique under Efficient Replacement Policy & it is observed that LRU partitioning perform better than Half-and-Half for some work load & perform equal for few workloads, but for most of the workload our new technique ECP outperform the best performing scheme out of other two conventional scheme. On average the ECP improves performance by 6.75% on conventional scheme under Efficient Replacement Policy.

Figure 4 shows percentage hit for each workload for all partitioning technique under LRU & ERP replacement policy. It is observed that Efficient replacement policy improves the performance for Half-and-Half partitioning technique by 3% on LRU replacement policy. For LRU partitioning Technique by 1.50% on LRU Replacement and new scheme Efficient cache partitioning Technique by3.50% on LRU Replacement.

V. RELATED WORK

Cho and Jin et al.[1], who proposed software-based mechanism for L2 cache partitioning based on physical page allocation. However, the major focus of their work is on how to distribute data in a Non-Uniform Cache Architecture (NUCA) to minimize overall data access latencies. However, they do not concentrate on the problem of uncontrolled contention on a shared L2 cache.

David Tam et al. [2], demonstrated a software-based cache partitioning mechanism and shown some of the potential gains in a multiprogrammed computing environment, which allows for flexible management of the shared L2 cache resource. This work neither supports the dynamic determination of optimal partitions nor dynamically adjusts the number of partitions.

Stone et al. [3] investigated optimal (static) partitioning of cache resources between multiple applications, when the information about change in misses for varying cache size is available for each of the competing applications. However, such information is non-trivial to obtain dynamically for all applications, as it is dependent on the input set of the application.

Suh et al. [4] described dynamic partitioning of shared cache to measure utility for each application by counting the hits to the recency position in the cache and used way partitioning to enforce partitioning decisions. The problem with way partitioning is that it requires core-identifying bits with each cache entry, which requires changing the structure of the tag-store entry. Way partitioning also requires that the associativity of the cache be increased to partition the cache among a large number of applications.

Qureshi et al. [5] proposed the cache monitoring circuits outside the cache so that the information computed by one application is not polluted by other concurrently executing applications. They provide a set sampling based utility monitoring circuit that requires storage overhead of 2KB per core and used way partitioning to enforce partitioning decisions. TADIP-F is better able to respond to workloads that have working sets greater than the cache size while UCP does not.

Chang et al. [6] used time slicing as a means of doing cache partitioning so that each application is guaranteed cache resources for a certain time quantum. Their scheme is still susceptible to thrashing when the working set of the application is greater than the cache size.

Suh et al. [7] described a way of partitioning a cache for multithreaded systems by estimating the best partition sizes. They counted the hits in the LRU position of the cache to predict the number of extra misses that would occur if the cache size were decreased. A heuristic used this number combined with the number of hits in the second LRU position to estimate the number of cache misses that are avoided if the cache size is increased.

Dybdahl et al. [8] presented the method which adjusts the size of the cache partitions within a shared cache, work did not consider a shared partition with variable size, nor did they look at combining private and shared caches.

Kim et al. [9] presented cache partitioning in shared cache for a two-core CMP where a trial and fail algorithm was applied. Trial and fail as a partitioning method does not scale well with increasing number of cores since the solution space grows fast.

Z. Chishti et al. [10] described spilling evicted cache blocks to a neighboring cache. They did not consider putting constraints on the sharing or methods for protection from pollution. No mechanism was described for optimizing partition sizes.

Chiou et al. [11] suggested a mechanism for protecting cache blocks within a set. Their proposal was to control which blocks that can be replaced in a set by software, in order to reduce conflicts and pollution. The scheme was intended for a multi-threaded core with a single cache.

Dybdahl et al. [12] presented a approach in which the amount of cache space that can be shared among the cores is controlled dynamically, as well as uncontrolled sharing of resources is also control effectively. The adaptive scheme estimates, continuously, the effect of increasing/ decreasing the shared partition size on the overall performance. Paper describes NUCA organization in which blocks in a local partition can spill over to neighbor core partitions. Approach suffers from pollution and harmonic mean problem.

DimitrisKaseridis et al. [13] proposed a dynamic partitioning strategy based on realistic last level cache designs of CMP processors. Proposed scheme provides on average a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static equally partitioned (private) caches. This work highlights the problem of sharing the last level of cache in CMP systems and motivates the need for low overhead, workload feedback-based hardware/software mechanisms that can scale with the number of cores, for monitoring and controlling the L2 cache capacity partitioning

VI. CONCLUSION

The performance of Chip multiprocessor systems depend on an effective cache system. We have improving the cache system by adapting the cache usage per core to its needs by protecting its most recently used data in the last-level cache. Results show that the new partitioning scheme has higher Performance under efficient cache replacement policy rather than conventional LRU replacement policy which are almost all used in all partitioning technique. As we have divide L2 cache in to private and shared cache it reduce average memory access latencyas hits to the private cache are faster than in a shared cache due to its smaller size. Each core is protected from pollution by the other cores and is given a cache block as per the benefit rather than demand which minimizes the total number of cache misses for all cores.

REFERENCE

1. S. Cho and L. Jin, “Managing Distributed, Shared L2 Caches through OS-level Page Allocation”, Proceedings of the Workshop on Memory System Performance and Correctness, 2006.
2. David Tam, Reza Azimi, LivioSoares, and Michael Stumm, “Managing Shared L2 Caches on Multicore Systemsin Software”, Workshop on the Interaction between Operating Systems and Computer Architecture, 2007.
3. H. S. Stone, J. Turek, and J. L. Wolf.,“Optimal Partitioning of Cache Memory” IEEE Transactions on Computers, 41(9):1054–1068, 1992.
4. G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic Partitioning of Shared Cache Memory” Journal of Supercomputing, 28(1):7–26, 2004.
5. M. K. Qureshi and Y. Patt,“Utility Based Cache Partitioning: ALow Overhead High-Performance Runtime Mechanism to Partition Shared Caches”,The $39^{'th}$ Annual IEEE/ACM International Symposium on Microarchitecture,MICRO'06
6. J. Chang and G. S. Sohi,“Cooperative Cache Partitioning for Chip Multiprocessors”,Proceeding of 22^{nd} Annual International Conference on Supercomputing, ICS-21, 2007.

7. G. Suh, S. Devadas, and L. Rudolph,“Dynamic Cache Partitioning for Simultaneous Multithreading Systems”, International Conference On Parallel and Distributed Computing Systems, 2002.

8. H. Dybdahl, P. Stenstrom, and L. Natvig“A Cache Partitioning Aware Replacement Policy for Chip Multiprocessors”, In 13^{th} International Conference High Performance Computing, HiPC, 2006.

9. C. Kim, D. Burger, and S. W. Keckler,“Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches”, IEEE Micro 2004, 23(6): 99-107.

10. Z.Chishti,M.D.Powell, and T. N. Vijaykumar,“Optimizing Replication Communication and Capacity Allocationin CMPs”, 32^{nd} Annual International Symposium on Computer Architecture, ISCA, 2005, pp: 357-368.

11. D.Chiou,P.Jain, S. Devadas, and L. Rudolph,“Dynamic Cache Partitioning via Columnisation”, Proceedingsof the 37^{th} ConferenceonDesign Automation, Los Angeles, June5-9, 2000,ACM, 2000.

12. Haakon Dybda, Perstenstrom, “An Adaptive Shared/Private NUCA Cache Partitioning Schemefor Chip Multiprocessors”, IEEE 13^{th} International Symposium onHigh Performance Computer Architecture, 2007, pp: 2 – 12.

13. DimitrisKaseridis,Jeffrey Stuechelixand Lizy K.John,“Bank-aware Dynamic Cache Partitioning for Multicore Architectures 38^{th} International Conference on Parallel Processing 2009