Sharing and Hit based Prioritizing Replacement Algorithm for Multi-Threaded Applications

Muthukumar S
Associate Professor, Department of CSE,
Sri Venkateshwara College of Engineering,
Tamil Nadu, India.

Jawahar P.K
Professor, Department of ECE,
BS Abdur Rahman University,
Tamil Nadu, India

ABSTRACT
Cache replacement techniques like LRU, MRU etc. that are currently being deployed across multi-core architecture platforms, try to classify elements purely based on the number of hits they receive during their stay in the cache. In multi-threaded applications data can be shared by multiple threads (which might run on the same core or across different cores). Such data needs to be given more priority when compared to private data because miss on those data items may stall the functioning of multiple threads resulting in performance bottleneck. Since the traditional algorithms mentioned above do not possess this additional capability, they might lead to sub-optimal performance for most of the current multi-threaded applications. To address this limitation, our paper proposes a Sharing and Hit Based Prioritizing (SHP) replacement strategy that takes the sharing status of the data elements into consideration while making replacement decisions. Every cache element is associated with a 'Sharing Degree' which indicates the extent to which the element is shared based on the number of threads that try to access that element. There are four degrees of sharing namely – Not shared (or private), lightly shared, heavily shared and very heavily shared. Combining the sharing degree along with the number of hits received by the element, we embark on a priority based on which the replacement decisions are made. Evaluation results obtained using multi-threaded workloads derived from the PARSEC benchmark suite shows an average improvement of 4% to 5% in the overall hit rate when compared to LRU algorithm.

General Terms
Multi-Core Architecture, Cache Memory, Multi-Threaded Applications.

Keywords
Sharing, Cache, Replacement, Hits, Threads.

1. INTRODUCTION
Modern day applications possess high computational and throughput requirements. To expedite their execution speed, these applications spawn multiple threads which run in parallel. Each thread has its own execution context and is assigned to perform a particular task. In a multi-core architecture, these threads can run on the same core or on different cores. Generally any multi-core architecture will have the following cache memory hierarchy- a private L1 cache for each core and a relatively larger L2 cache which is shared by all the cores.

When threads run across different cores, the activity in L2 cache tend to shoot up as multiple threads try to store and retrieve shared data. At this point there are many challenges that need to be dealt with. Cache coherence has to be maintained, the replacement decisions made need to be judicious and the available cache space must be utilized efficiently. It is to be noted that when there is a miss on a data item which is shared by multiple threads, the execution of many threads gets stalled. This is because when the first thread, which had encountered a miss, is attempting to fetch the data from the next level of memory, any subsequent thread which comes looking for the same data will result in miss.So it becomes imperative to handle shared data with more care compared to other data items. Traditional replacement algorithms like LRU, MRU etc do not possess this capability. They classify elements based on when they will be required by the processor but do not check for the status of the cache block, i.e. whether it is shared or private at any point of time. Also when there are more than thousands of threads running in parallel, it may not be very useful in tagging the block simply as 'shared' or 'private'. In those cases, additional information about their shared status can prove handy.

Thus in this work we have come up with a novel counter based cache replacement strategy that associates a 'sharing degree' with every cache block. This degree specifies a range which includes – not shared (or private), lightly shared, heavily shared and very heavily shared. This information combined with the number of hits received by the element during its stay in the cache is used to produce a priority for that element based on which judicious replacement decisions are taken.

Rest of the paper is organized as follows: section 2 looks into the related work that was done in this field, section 3 explains the working of our replacement technique in detail, sections 4 and 5 describes the experimental setup and analyses the obtained results respectively and finally section 6 summarizes the paper followed by the list of references.

2. RELATED WORK
Shared Last Level Cache (LLC) is accessed by multiple cores. So it is important to have a good, efficient replacement algorithm running over it. When a miss is encountered here, the resulting overhead can be higher compared to other cache levels. Many works [2,3,5,6,8,9] have emerged in recent times which strive to improve the performance at LLC.

A method which was proposed by Mainak Chaudhuri et al [8] discusses on how the activities occurring in the inner levels of the cache can be used to make replacement decisions in the LLC. Here activities refer to the pattern of hits and misses encountered. But communicating such information across various levels of cache frequently can cause significant overhead.

Fazal Hameed et al [9] proposed a dynamic cache management scheme targeted towards LLCs. But it does not

attach any importance to shared data. Elimination of the dead-lines (or) the lines which will never be accessed by the processor in the near future [3,4,11] can greatly enhance the performance of cache memory. Livio Soares et al.[3] have proposed a technique to get rid of the dead-lines but they work from OS level and might burden the OS over a period of time. Counter based replacement technique [4] tries to predict dead-lines well in advance and choose them as replacement victims.

Efficient cache partitioning can help in improving cache performance. The work by Konstantinos Nikas et al [7] suggests a dynamic cache partitioning technique using bloom filters and counters. Since every core is allocated an array of bloom filters and counters, the hardware complexity in this method can shoot up as the number of cores increases. Phase Change Memory (PCM) was suggested as an alternative to the traditional DRAM and techniques have been proposed to improve the performance of the LLC in PCM [10] but the drawback of PCM is that the writes are much slower compared to DRAM.

Almost all the techniques discussed above do not attach importance to data that is shared by multiple threads. Hence in this work we focus on designing a novel counter based replacement algorithm for shared LLC in a CMP environment To bring the status of the cache block into picture, we allocate a 'sharing degree' counter with each and every cache block to classify it into any one of the four groups. Number of hits received by the data item is also an important factor to be considered. This number is combined along with the sharing degree to arrive at a priority for the cache block which assists in making replacement decisions.

3. COUNTER BASED PRIORITIZING APPROACH

Every cache block is associated with a 2-bit counter. This counter is called as the Sharing Degree Counter or just SD counter. Table 1 shows all the four possible values this counter can contain. For demonstration purposes, the maximum number of threads that can be created is set to 10. Based on the number of threads that try to access a data item, we classify it into any one of the sharing categories as shown in the table. The mapping policy employed at the cache is taken as set associative mapping [1].

To collect the sharing status of the cache blocks, it is essential to have an efficient data structure in place to track the number of threads that accesses the data item. For this purpose we have a filter which is referred to as the Thread Tracker filter (or just TT filter). It is a flexible dynamic software based array that gets created and is associated with every cache block during run time.

When a thread tries to access a data item, a search is conducted in the TT filter to check if the thread id is already present in it. If not, then the id is stored in the corresponding cache block's TT filter. Size of the filter expands as and when a thread id gets added to it. Once a cache block is about to be evicted, the memory allocated for the associated TT filter is freed. At any point of time, based on the number of threads that are found in the TT filter, the sharing degree counter is populated for every cache block.

Table 1. Sharing degree values and their descriptions

Number of Accessing Threads	Sharing Degree Counter Value	Nature of Sharing
1	0	Private/Not Shared
2-3	1	Lightly Shared
4-7	2	Heavily Shared
8-10	3	Very Heavily Shared

3.1 Priority Computation

As discussed in the earlier sections, the sharing status of the blocks alone cannot be used to make replacement decisions. Justification for which goes as follows:

For example, there are two data items in the cache. One is shared by (say) 10 threads, so it will be having the highest sharing degree counter value of 3. The other data item is private to one thread. But this data item is used much more frequently by the thread than the shared one. Hence it receives huge number of hits (say 50) whereas the shared data item has earned only 20 hits. Though the first element is shared by more number of threads, it is something that is not required much over a period of time. So when a replacement decision has to be made, it is this element which needs to be picked up as the victim rather than the private data. This scenario indicates that the number if hits garnered by the element is also an important factor to consider when performing a replacement. Hence we have arrived at a priority computation formula that not only gives more weightage to sharing nature but also attaches importance to the hit count.

*Priority = (Sharing Degree + 1) * (Hit Count / 2)*

Priority is set to '0' initially for all the blocks. Sharing degree is incremented by 1 before computing the product as it can be seen that the minimum value sharing degree can hold is set to 0. When the product is computed, priority can also result in 0 which is not desired (apart from the first time). Replacement, insertion and deletion form the heart of any replacement algorithm. Each phase of our algorithm is explained in detail in the subsequent sections.

3.2 Replacement

When the cache becomes full, replacement has to be made to pave way for new incoming data items. SHP makes replacement decisions based on the computed priority values. Elements are evicted in the increasing order of their priority. The element with the least priority in the list is chosen as the victim. If more than one element has the same least priority, then the one which is encountered first while scanning the cache is taken as the victim as a tie-breaking mechanism. It is also essential to ensure that stale data do not pollute the cache for longer periods of time. For this purpose, every time a victim is found, the hit counter of all the other elements in the cache is decremented by '1' and their corresponding priorities are re-computed. If any element remains unreferenced for a long period of time, its priority will gradually decrease and the element will eventually be flushed out of the cache.

3.3 Insertion

After evicting the victim, the new data item needs to be inserted into the cache. Since initially all the blocks would contain invalid data and their corresponding priority values will be '0', the incoming blocks' priority must be set to some other value other than '0'. It cannot be given a higher priority

value since we are not sure about its sharing nature and the amount of hits it might receive in the future. So we have chosen a random priority value of '10'. Sharing degree counter is set to '0' (to indicate that the incoming block is currently not shared by any other threads).

3.4 Promotion

When a cache hit happens, the hit counter of the corresponding block is incremented by '1' and the priority is re-computed. Also the accessing thread id is compared against the ids which are already present in the TT filter and if it is not present there, it is added into the TT filter. Sharing degree counter is adjusted accordingly.

3.5 Scalability

For illustration purpose, the number of threads is taken to be 10. In real time applications, this number can be quite high. Irrespective of the number of threads under execution, the sharing degree can be set proportionally similar to the way it has been done with 10 threads. For example if there are (say) 1000 threads, then if 200-300 threads share a data item it is lightly shared and if 400-700 threads share a data item it can be regarded as heavily shared and so on.

4. EXPERIMENTAL SETUP

An open-source, full system simulator called Gem5[12] which is capable of simulating a variety of Instruction Set Architectures (ISAs) has been chosen to evaluate our method. The cache and the processor configuration go as follows: Alpha ISA has been chosen with 2 cores which operate at 2 GHz clock frequency. Supported cache levels include a private L1 cache which is further sub-divided into instruction and data cache and a relatively larger L2 cache which is shared between the available cores. The size of L1 and L2 cache are set to 64 kB and 2 MB respectively. Line size for both the caches is 64B. L1 cache is 2-way associative and L2 cache is 8-way associative. SHP algorithm is applied at L2 whereas L1 runs LRU algorithm. Seven versatile workloads have been picked from the Princeton application repository for shared-memory computers (PARSEC), [13, 14] a benchmark suite that comprises numerous large scale commercial multi-threaded workloads targeted towards CMP, to evaluate our method. Table 2 highlights the key characteristics of all the PARSEC benchmarks used.

5. RESULTS AND DISCUSSIONS

The main parameters involved in measuring the performance of any memory system include the data hits and misses. Overall number of hits obtained at L2 cache for our method compared to LRU is shown in the graph in Figure. 1. In every figure, the y-axis indicates the parameter under scrutiny and x-axis indicates the various benchmarks. In Figure.1 *ferret* benchmark has shown the maximum improvement. On an average, a 5% percent improvement in the overall number of hits can be observed across the given benchmarks.

Table 2. Key characteristics of PARSEC benchmarks

Program	Application Domain	Working Set
Blackscholes	Financial Analysis	Small
Canneal	Computer Vision	Medium
Dedup	Enterprise Storage	Unbounded
Ferret	Similarity Search	Unbounded
Swaptions	Financial Analysis	Medium
Vips	Media Processing	Medium
X264	Media Processing	Medium

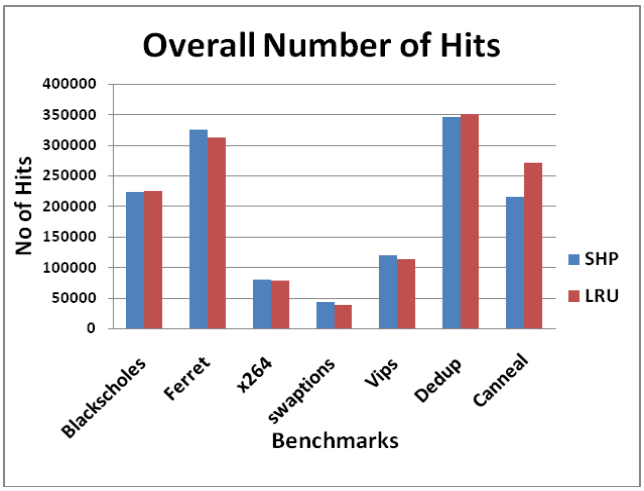

Figure 1: Overall number of hits at L2 cache

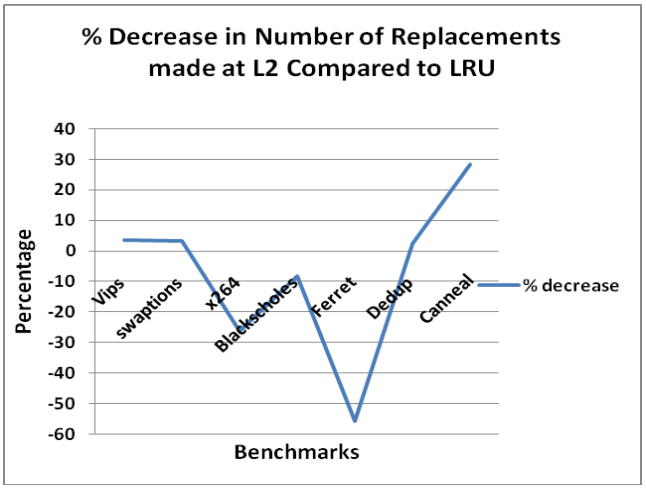

Figure 2: Percentage decrease in number of replacements made at L2 compared to LRU

Figure. 2 shows the overall number of replacements made at L2 for SHP and LRU. The more the number of replacements made, the more will be the overhead involved. So it is always desirable to keep this parameter as low as possible. In our method, the average number of replacements made at L2 has decreased by almost 10 % compared to LRU. Figure. 3 shows the miss rate measured across the given workloads. Miss rate is computed from the overall number of misses and the overall number of accesses.

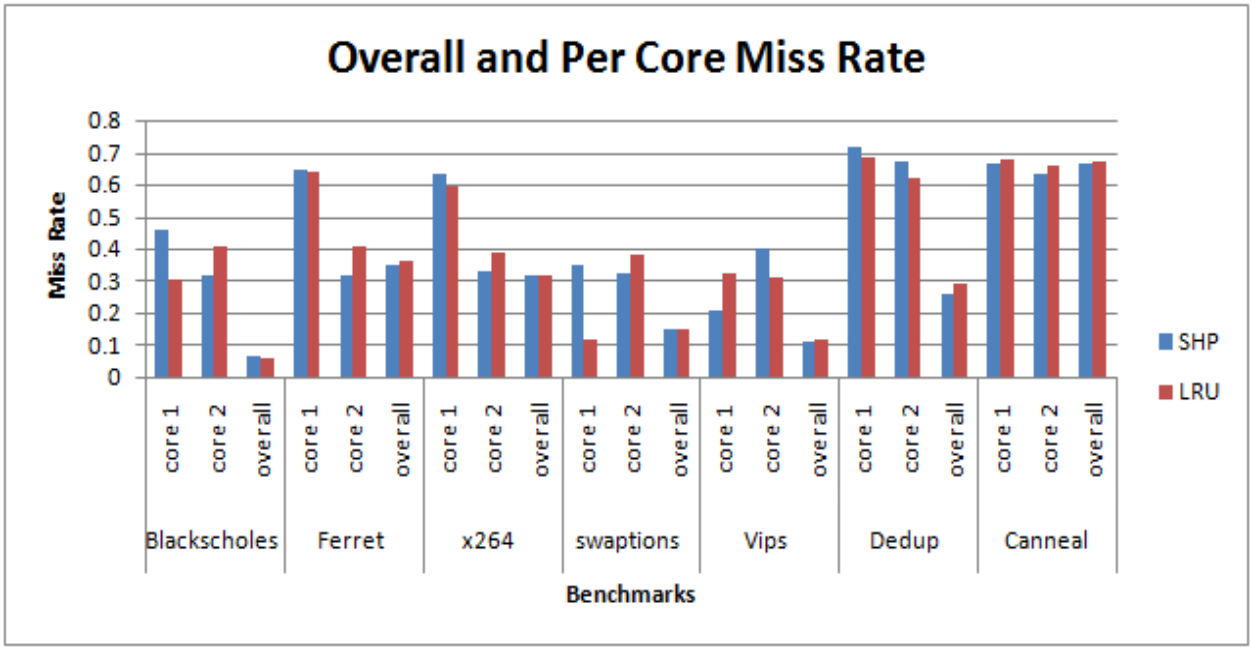

Figure 3: Overall miss rate and core-wise miss rate at L2 cache

To have an impressive performance, miss rate needs to be kept as low as possible. Figure.3 shows that majority of the benchmarks have shown marginal improvement in the miss rate when compared to LRU.

6. CONCLUSION

Shared data plays a crucial role in determining the performance of cache memory systems, especially in a multi-threaded environment. Conventional LRU approach does not attach importance to such data and hence in this work we have come up with a novel counter based prioritizing algorithm.

- Every cache block is associated with a 2-bit sharing degree counter which iterates from 0 to 3.
- A dynamic software based TT filter is associated with every block to keep track of the threads that are accessing that block.
- A hit counter is used to keep track of the hits received by the data item.
- Values of the sharing degree and the hit counters are used to compute a priority for each cache block.
- This priority is then used to make judicious replacement decisions.

Evaluation results have shown an average improvement of up to 5% in the overall number of hits when compared to the traditional LRU approach.

7. REFERENCES

[1] John L. Henessey, David A. Patterson. 2006. Computer Architecture: A Quantitative Approach, Fourth Edition, Elsevier Publications.

[2] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Joel Emer, "Adaptive Insertion Policies for Managing Shared Caches", ACM Parallel Architectures and Compilation Techniques (PACT), Oct. 2008, p.208-219.

[3] Livio Soares, David Tam, Michael Stumm, "Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-level, Software-Only Pollute Buffer", 41st Annual IEEE/ACM International Symposium on Microarchitecture, 2008, p.258-269.

[4] Mazen Kharbutli, Yan Solihin, "Counter Based Cache Replacement and Bypassing Algorithms", IEEE Transactions on Computers, Vol. 57, Issue. 4, April 2008, p.433-447.

[5] Carole-Jean Wu, Margaret Martonosi, "Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches", ACM Transactions on Architecture and Code Optimization, Vol. 8, No. 1, Article 3, April 2011.

[6] Shekhar Srikantaiah, Mahmut Kandemir, Mary Jane Irwin, "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors", ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS), Vol. 36, Issue. 1, March 2008, p.135-144.

[7] Konstantinos Nikas. Matthew Horsnell. Jim Garside. 2008. An Adaptive Bloom Filter Cache Partitioning Scheme for Multi-Core Architectures. In Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation, p.25-32.

[8] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Srinivas Subramoney, Joseph Nuzman, "Introducing Hierarchy-Awareness in Replacement and Bypass Algorithms for Last-Level Caches", ACM Parallel Architectures and Compilation Techniques (PACT), Sep. 2012, p.293-304.

[9] Fazal Hameed. Bauer L. and Henkel J. 2012. Dynamic Cache Management in Multi-Core Architectures through Runtime Adaptation. In Proceedings of Design Automation & Test in Europe Conference & Exhibition (DATE), p.485-490.

[10] Miao Zhou, Yu Du, Bruce Chilers, Rami Melham, Daniel Mosse, "Writeback-Aware Partitioning and Replacement for Last-Level Caches in Phase Change Main Memory Systems", ACM Transactions on Architecture and Code Optimization, Vol. 8, No. 4, Article 53, Jan. 2012.

[11] Haiming Liu, Michael Ferdman, Jaehyuk Huh, Doug Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency", 41st Annual IEEE/ACM International Symposium on Microarchitecture, Vol. 1, Issue. 12, 2008, p.222-233.

[12] N. Binkert et al. "The gem5 simulator", SIGARCH Computer. Architecture New, Vol. 39, Issue. 2, May 2011, p.1-7.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", Princeton University Technical Report, TR-811-08, Jan. 2008.

[14] M. Gebhart et al., "Running PARSEC 2.1 on M5", University of Texas at Austin, Department of Computer Science, Technical Report, TR-09-32, Oct. 2009.