# Improving Cache Performance by Exploiting Read-Write Disparity for Exclusive Caching


Aurosmita Khansama
*Dept. of Computer Science and Engineering*
*Texas A&M University*
College Station, USA
aurosmita.k@tamu.edu
UIN 230002654

Hong-Jie Chen
*Dept. of Computer Science and Engineering*
*Texas A&M University*
College Station, USA
elvis.hongjie.chen@tamu.edu
UIN 430000712

Pei Chen
*Dept. of Computer Science and Engineering*
*Texas A&M University*
College Station, USA
chenpei@tamu.edu
UIN 130005135

Rizu Jain
*Dept. of Computer Science and Engineering*
*Texas A&M University*
College Station, USA
rizujain@tamu.edu
UIN 430000753



***Abstract*— A major bottleneck in the memory wall impacting the performance gap between the processor and memory is the memory requests that takes hundreds of cycles for memory access. This limitation tends to be an inspiration to improve further on cache design and replacement policies. Traditionally, a cache is designed such that when the blocks in a set are evicted, it does not distinguish the write lines from the read lines. A general observation is seen that the cache lines that serve the store instructions as not as critical as the cache lines that serve the load instructions. The traditional method does not address the difference between the latencies of read misses and write misses. The primary idea in the paper is to implement a policy i.e. Read Write Partitioning (RWP) that will target to minimize the read misses. This shall be done by dynamically partitioning the cache into dirty and clean partitions. The size of the partition is dependent on the predicted read requests. This will account for the disparity between write and read lines and reduce the overall read misses. As a result, it provides slightly better results over the baseline LRU cache replacement policy.**



***Keywords—Dynamic Insertion Policy, Read Write Partitioning, Cache Replacement Policy, Exclusive Caches***


## I. INTRODUCTION

In the processor design, the latency of the read requests from loads is treated with great attention while the write requests from stores are buffered for some cycles before the commit stage of the pipeline. The same priority is not given in the traditional memory design. Hence, although the read request latency has higher criticality than the write latency, the treatment of load and stores are the same.

The idea implemented in the paper [1] comes from the above-mentioned limitation of the traditional memory design. We should implement the cache replacement policies such that critical read requests are favoured over less critical write requests. Figure 1 depicts the read miss that shall stall the processor. Since write misses are buffered, in Figure 2 a write miss does not stall the processor. Another read request will stall the processor again in Figure 3.


*Figure 1: Miss at Read A*
*Source: Adapted from [1]*

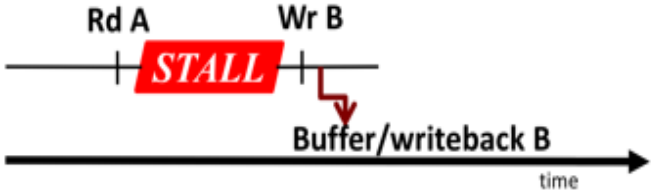


*Figure 2: Miss at Write B*
*Source: Adapted from [1]*

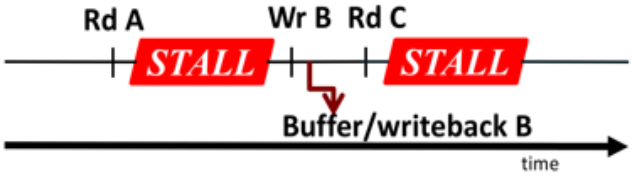


*Figure 3: Miss at Read C*
*Source: Adapted from [1]*

The following example describes the baseline LRU case and the proposed idea in the paper [1] which can limit these stalls and improve performance. Consider a program loop which shall access 4 lines in the cache A, B, C and D.


*Figure 4: Loop touching 4 caches lines*
*Source: Adapted from [1]*

In the baseline LRU policy, the line that has been unused for the longest time will be evicted out of the supposed cache that only contains three lines in this particular example.

Number of lines in cache = 3

Number of line accesses = 4

Read A will be a cache miss and it will cause a stall. [Figure 5]


*Figure 5: Miss at Read A*
*Source: Adapted from [1]*

Next, write B will also be a miss but no stalls in the processor since it is buffered. [Figure 6]


*Figure 6: Miss at Write B, No stall*
*Source: Adapted from [1]*

Next another access of B which is a hit since it is already stored in the cache line. [Figure 7]


*Figure 7: Hit at Read B*
*Source: Adapted from [1]*

When C is accessed through a write it will result in a miss but no stall. [Figure 8,9]


*Figure 8: Miss at Write C, no stall.*
*Source: Adapted from [1]*


*Figure 9: After Write C*
*Source: Adapted from [1]*

In the next read to D instruction, the processor needs to stall because of a miss here because of the eviction of D caused by C. [Figure 10]


*Figure 10: Miss at Read D*
*Source: Adapted from [1]*

In the above case for LRU policy, we see that the total number of stalls in the processor pipeline is two per iteration. [Figure 11]

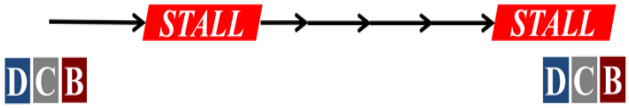


*Figure 11: Cache Lines Status in LRU*
*Source: Adapted from [1]*

In the proposed idea, we evict dirty lines for write requests since the read requests are critical and should result into a processor stall.

Consider that the first iteration of the loop has already been performed once. Access to A will be a hit because its read request is protected from the read request to A in the previous iteration. [Figure 12]

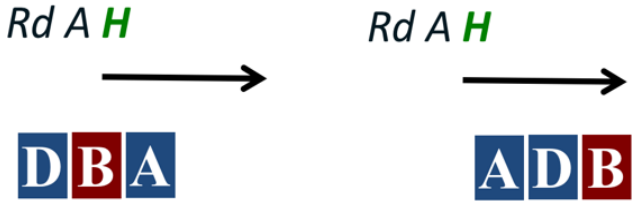


*Figure 12: Hit at Read A*
*Source: Adapted from [1]*

There will be two consecutive hits for Write to B and Read from B. [Figure 13,14]

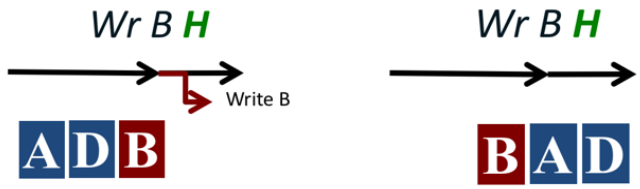


*Figure 13: Hit at Write B*
*Source: Adapted from [1]*

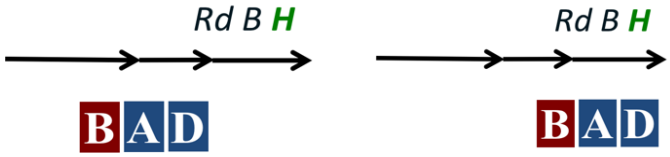


*Figure 14: Hit at Read B*
*Source: Adapted from [1]*

The next write to C will be a miss as it is not present in the cache. [Figure 15]

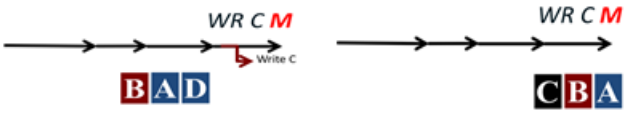


*Figure 15: Miss at Write C*
*Source: Adapted from [1]*

Next, when the read request to D is made, it will result into a miss. [Figure 16] RWP functions here: Instead of replacing the LRU block, we will replace that block which does not do any read accesses. Hence C will be evicted.

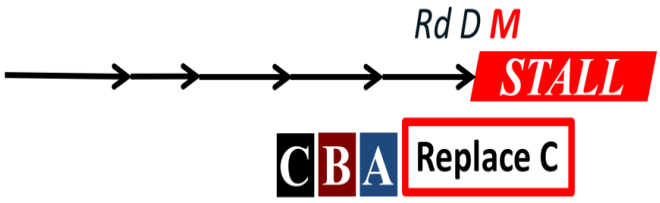


*Figure 16: Miss at Read D*
*Source: Adapted from [1]*

Overall only one stall occurs since A is protected and a miss resulting from read A is eliminated. So in RWP only one stall per instruction occurs. [Figure 17]

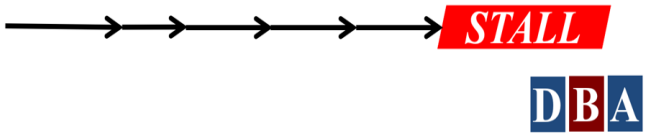


*Figure 17: Cache Line Status in RWP*
*Source: Adapted from [1]*

## II. Related Work

There has been several prior work which concentrated on identifying and making use of the locality of the cache working set by optimising the cache replacement and insertion policies. For instance, the work may identify the working set that were mostly reused. The work didn't exploit the criticality of the members of the working set.

Some research was also done as reflected by Piquet et al [6] to use the PC of some memory reference instructions that request cache lines to predict the extent of reuse as exhibited by the memory requests. Read Reference Predictor (RRP) in [1] was built upon this idea to measure the reuse of read lines. The primary observation that was sought from this implementation was if the cache lines allocated by an instruction are subsequently used for loads.

Some implementation issues were observed in its complexity although the idea of RWP is better implemented in RRP. Some additional recording of the PC information to the last level cache for write-backs are needed for accurate results that would increase the hardware bandwidth.

## III. Read Write Partitioning

### A. *RWP Framework*

RWP separates cache lines to read part and write part for each set of cache. By partitioning, RWP can easily favour read lines over write-only lines by choosing line from certain part as an evicted block in replacement policy. However, to separate cache lines, cache needs to know the whole instructions in trace file, which is not possible. To solve the issue, RWP managed to predict future instruction type by the past instruction types. The easiest way to predict the future instruction types is to predict clean lines to be read and dirty lines to be written. However, most of the time, that's not the case.

To address the problem, RWP separate cache lines into dirty part and clean part. Dirty part includes all lines that have been written to, while clean part includes all lines that have been read. On top of that, RWP predicts a write-only part, which is a subset of the dirty part and is not expected to be read in the next instruction. To maximize the number of read hit, RWP attempts to minimize the write-only part in the cache. RWP uses dynamic partitioning to do the prediction.

After we get the prediction, there are three cases, shown in the below figure, we should take into consideration when we try to assign a block to evict:

- Number of actual dirty lines greater than predicted size of dirty lines: RWP chooses the block from the dirty part using LRU method.
- Number of actual dirty lines smaller than predicted size of dirty lines: RWP chooses the block from the clean part using LRU method.
- Number of actual dirty lines equal to predicted size of dirty lines: It depends on the instruction type. If it is a read, than RWP chooses the evicted block from the clean part. If it is a write, than RWP chooses the evicted block from the dirty part.

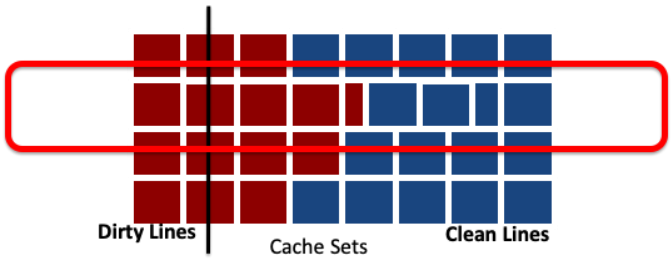


*Figure 18: Partitioning based on dirty and clean lines*
*Source: Adapted from [1]*

### B. *Dynamic Partitioning*

- Since different workloads have different degree of read/write reuse of cache, RWP dynamically adjusts the predicted partitioning by the instruction type history. Each instruction access triggers a RWP evaluation, and might change the prediction.
- To estimate the partition size, RWP extends a subset with two shadow directories, one for clean lines and the other for dirty lines. When a read miss occurs, RWP allocates a line to the clean lines shadow directory. When a write miss occurs, RWP allocates a line to the dirty lines shadow directory. In addition, a write hit on a clean line also causes a line be moved from clean shadow directory to dirty shadow directory.
- RWP also maintains two global age hit counters, one for clean lines and the other for dirty lines. The two counters increment when a hit occurs at corresponding LRU position in the shadow directory. By comparing the dirty and clean counter's value, RWP can predict the number of hits that will occurs in the future and adjust the line of prediction.

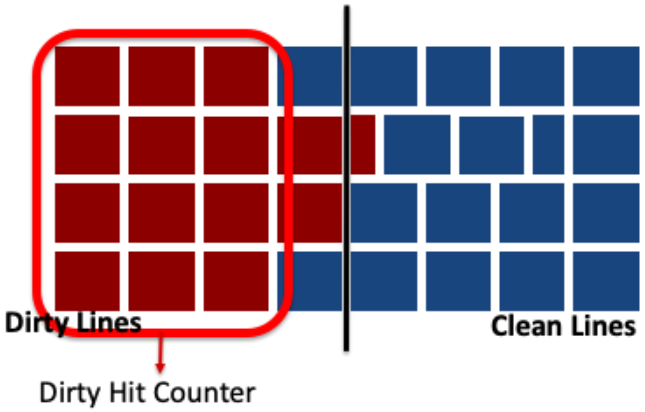


*Figure 19: Dirty Hit Counter*
*Source: Adapted from [1]*

## IV. Evaluation Methodology

SPEC2006 is used as benchmark for this project. It has 29 benchmarks with 12 integer and 17 floating point programs. For this project we have considered 27 benchmarks out of which 10 are integers and 17 are floating point. For an effective analysis of the project, load and store instruction percentage is needed. This will show whether the benchmark is read intensive or write intensive. Load and store percentage for the given benchmarks was obtained from [5]. Figure 20 shows the percentage of load and store instructions for the benchmarks.

This shows the behaviour of the benchmarks that will help us understand the working of the read-write disparity policy.

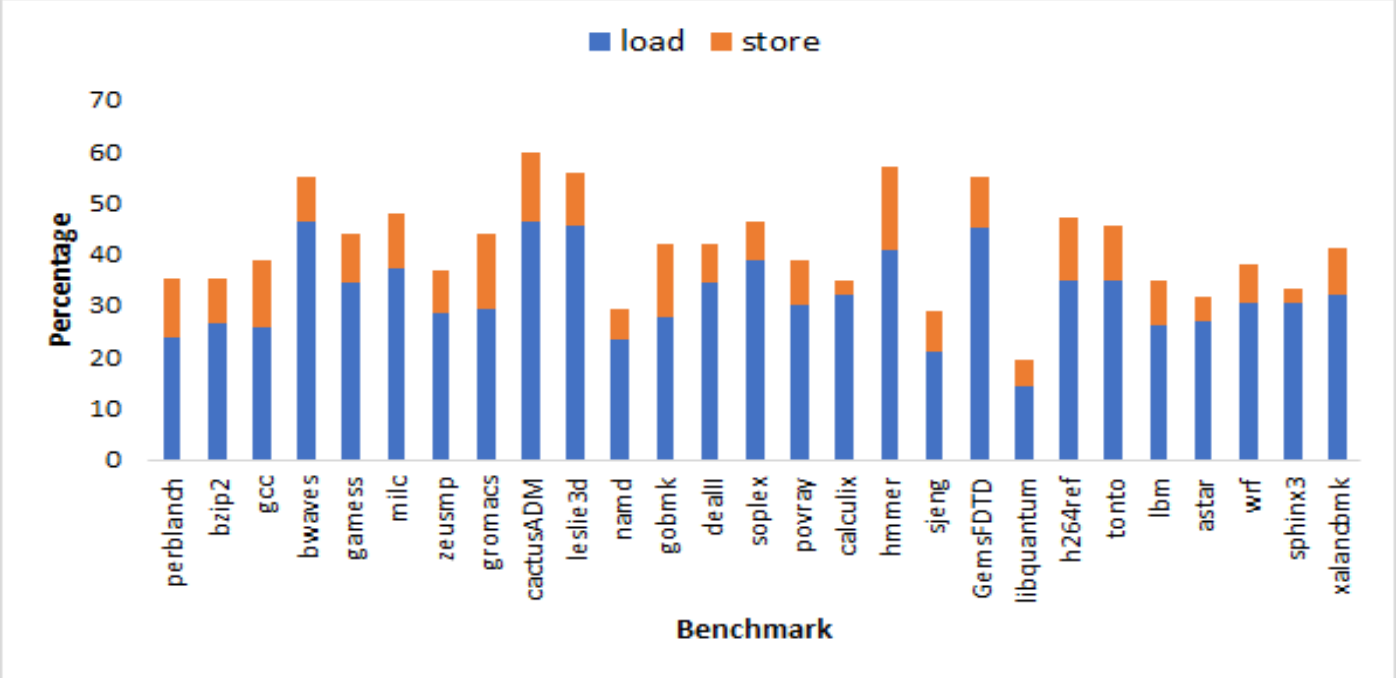


*Figure 20: Percentage of load and store instructions for various benchmark*

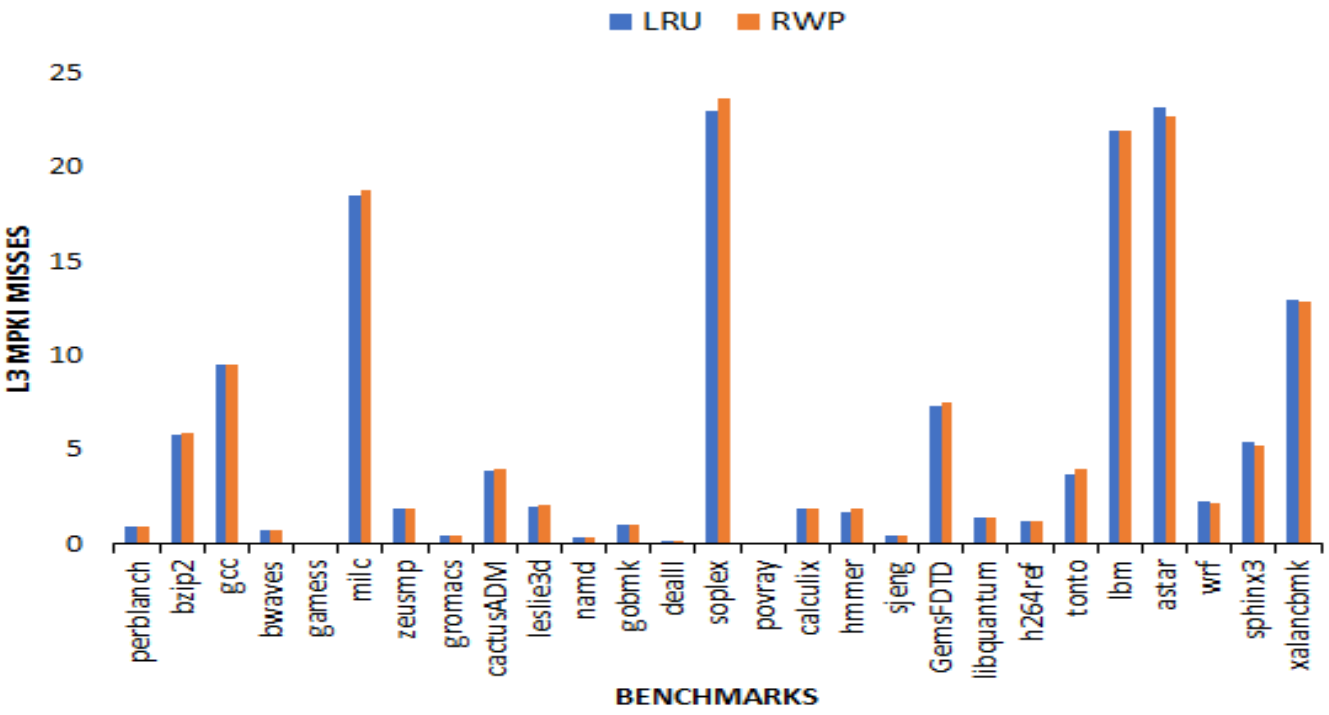


*Figure 21: Comparison of MPKI misses between LRU and RWP for various benchmark*

## V. PROFILING AND RESULTS

Cache miss occurs when a requested memory block is not found in a cache. This causes processing stalls and requires to fetch the data from lower-level cache levels or the main memory. Therefore, the cache miss rate is used as a major metric to determine cache performance. We have considered MPKI or misses per kilo instructions to compare RWP with LRU. Fig.21 illustrates that comparison.

- On an average, 2-4% reduction in mpki was found in few benchmarks such as perblanch, gcc, bwaves, gromacs, gobmk, dealII, h264ref, astar, wrf, sphinx3, and xalancbmk. The improved performance is due to RWP's superior capability to reduce read misses in these load intensive benchmarks[1].
- More specifically, a maximum reduction of 4.8% was noticed for sphinx3 benchmark, which is followed by 4.75% of dealll benchmark.

It was observed that maximum reduction in misses was found for benchmarks that are read intensive. Benchmark Sphinx3 has 90 percent load intensive instruction (with 30.4% load instructions and 3% store instructions) and thus there is 4.8% reduction in miss rate. Table 1 shows comparison of statistics of LRU and LRU with RWP replacement policies for the particular cache configuration and Sphinx3 benchmark.

However, for the rest of the benchmarks, RWP either show negligible improvements or slightly higher MPKI value. For example, with Soplex benchmark, RPW has 2.7% higher misses compared to LRU. It happens when RWP reduces the read misses at the cost of increasing write traffic. The increase in write traffic leads to an increase in write misses [1].

*Table 1: Table showing statistics for the best performing trace i.e. Sphinx3 benchmark*

| | LRU | RWP |
|---|---|---|
| **L3 instructions** | 599774938 | 599774938 |
| **L3 misses** | 3285899 | 3129060 |
| **L3 mpki** | 5.4786 | 5.2171 |
| **core 0** | 1.0737 IPC | 1.1078 IPC |
| **LLC invalidations** | 5481495 | 5416094 |

### A. *Rationale of Low Geometric Mean of Speedup for all the traces*

Overall, the implemented RWP showcases erratic behaviour when the geometric mean of speedup over LRU for all the traces are calculated. This is because of the valid result that RWP may not show performance improvement over LRU for all kinds of traces. It strictly depends on the characteristics of the benchmarks and the distribution of the load and store instructions.

## VI. CONCLUSION AND FUTURE WORK

In this project, qualitative analysis of the paper [1] was made which contributed in implementation of the Read Write Partitioning policy so proposed in the paper for exclusive

caches. Having taken advantage of the frequency of cache read requests on critical lines versus the frequency of cache writes, performance gap between processor and memory design can be bridged. This improve the overall system performance. For a single core system, it was observed that RWP with dynamic partitioning outperforms the baseline LRU policy by reducing the miss rate by more than 4 percent.

The scope of RWP policy can be extended to other cache configurations that perhaps target another optimisation. For example RRIP insertion policy will prohibit thrashing in cache.

Different applications may have different requirements in terms of write operations. As an instance, non-volatile memory requires that the write requests be treated critically. In this manner, RWP may exhibit a different reaction in such an environment. There may be substantial future work involved for the same.

## REFERENCES


[1] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutluy and D. A. Jimenez, "Improving cache performance using read-write partitioning," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, 2014, pp. 452-463.

[2] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely and J. Emer, “Adaptive insertion policies for high performance caching” ACM SIGARCH Computer Architecture News vol.35, no.2, pp. 381-391, 2007

[3] G. Keramidas, P. Petoumenos and S. Kaxiras, "Cache replacement based on reuse-distance prediction," 2007 25th International Conference on Computer Design, Lake Tahoe, CA, 2007, pp. 245-250.

[4] S. M. Khan, Y. Tian and D. A. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches," 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, 2010, pp. 175-186.

[5] Bird S., Phansalkar A., John L.K., Mericas A., Indukuru R., "Performance characterization of SPEC CPU benchmarks on Intel’s core microarchitecture based processor", 2007 SPEC Benchmark Workshop.

[6] T. Piquet et al. Exploiting single-usage for effective memory management. ACSAC, 2007

[7] Hennessy J. L. and Patterson D. A. (2017) Computer Architecture: A Quantitative Approach, Sixth Edition. Morgan Kaufmann.