ECE565 Project : Variable Way Cache

Patel Jhalak, Zaidy Aliasger, Hallymysore Sumukh, Nagaraj Vivek

***Abstract*—Intelligent design and management of secondary caches is a very important and highly researched area. The non-uniform distribution of memory accesses across different cache sets is one of the primary reasons for the reduced efficiency of set associative caches. The replacement policies generally do not take global use frequency of caches into account. This can lead to repeated evictions (replacements) from few cache sets while other underutilized entries continue to occupy the caches for longer times, resulting in poor hit rate and increased misses. The technique proposed by Qureshi et al [1] tries to tackle this problem by varying the associativity of a cache on a per-set basis in response to the demands of the program. The proposed design achieves the performance benefits of global replacement, by increasing the number of tag-store entries relative to the number of data lines, while maintaining the constant hit latency of a set-associative cache. We have tried to reproduce the Variable Way cache through demand based associativity via global replacement on gem5 simulator using both classic and ruby models, and have tabulated the results we observed for different benchmarks in SPEC CPU2006 suite.**

***Index Terms*—variable way, reuse replacement, demand based associativity, global replacement**

I. Introduction

Cache hierarchies play a crucial role in bridging the gap between processor speed and main-memory latency. The performance of a cache system directly depends on its success at storing data that will be needed by the program in the near future while discarding data that is either no longer needed or unlikely to be used soon. A cache manages this through use of replacement policy and prefetching algorithms. In a traditional set-associative cache, the number of entries visible to the replacement policy is limited to the number of ways in each set. On a miss, a victim is identified from one of the ways within the set. Set-associative cache cannot adapt associativity because data-lines are statically mapped to entries in the tag-store. In case of a cache miss, tag-store and data-store entries corresponding to the chosen victim are replaced. This combination of static mapping and local replacement results in reduced cache performance. The replacement policy could potentially select a better victim by considering the global access history of the cache rather than the localized set access history. This will result in an increase in number of hits and a reduction in the number of cache misses. This is particularly true because memory references in a program exhibit non-uniformity, causing some sets to be accessed heavily while other sets remain underutilized. One can achieve the lowest possible miss rate using a fully-associative cache with Belady's OPT replacement policy [2]. However, increased power consumption, latency and hardware costs make the implementation impractical. Also, without the knowledge of future memory accesses, it is impossible to get desired OPT replacement.

II. Proposed Solution

Varying the associativity of the cache provides better miss rate (lower miss rate) and global replacement allows fully-associative cache to choose the best possible victim every time. V-Way associative cache is interesting in the sense that it achieves performance benefit of global replacement while maintaining the constant hit latency of set-associative cache. In comparison to the traditional set-associative cache, in V-Way cache, associativity is varied on a per-set basis in response to the demands of the the application. The number of tag-store entries are increased relative to the data lines to accommodate data cache demand. The static one-to-one mapping of set-associative cache is replaced by newer dynamically mapped structures for the tag-store and data-store entries (using forward and reverse pointers). The maximum degree of associativity is limited to the number of tag-store entries in a set in the proposed structure, thereby achieving the constant hit latency of a set-associative cache. The reuse replacement policy that is used to select the victim has global information of the data usage frequency. The replacement policy uses 2-bit saturating counters to keep track of the reuse frequency. If an invalid tag store entry exists, these counters are used to find a global data victim for eviction, else LRU replacement policy is used to identify and evict the tag-store/data-store victim pair.

III. Implementation

We have implemented V-Way cache on both Ruby and Classic memory models. We have summarized our implementation and findings for both memory models in this section.

A. V-Way data structures

The proposed structure consists of a custom tag-store, a data-store and a reuse counter table(RCT). The custom tag store structure is twice (tag-to-data ratio = 2) the size of data store. It has double the number of sets as the data store but same number of ways to avoid greater increase in hit latency. The structure has a tag value, a pointer (forward pointer) and a time stamp. The forward pointer identifies the unique entry in data-store that the tag-store entry is mapped to (if any). Initially all the tag-store and data-store entries will be invalid. The RCT maintains a list of reuse counter values for each data-store entry. All the entries in the RCT are 2-bit saturating counters that are initialized to zero and can achieve a maximum value of three.

The set index is extracted from the requested address, and the tag-store set is searched for a matching tag value. In case of a hit, i) the forward pointer value is used to return the respective data entry block, ii) the corresponding count value in the RCT table is increased and iii) the time stamp of the tag-store entry is updated, which is required for finding victim

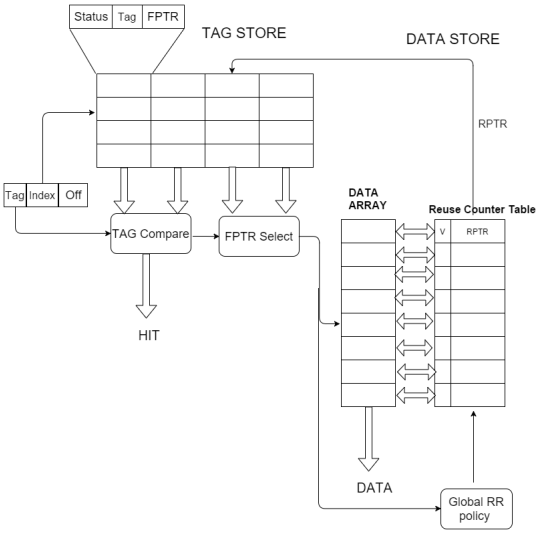

Figure 1. V-Way data structure

through LRU replacement policy. In case of a miss, there are two possible flows based on the whether an invalid entry is present in the tag-store set or not. In the former case, the victim is found by traversing the RCT table. The traversal is performed using the global replacement pointer. The pointer is initialized to point to the first data store entry. While finding the victim, the pointer is incremented till a zero RCT table entry is encountered. When the pointer reaches the end of the table, it is reset to the start. Each time the pointer is incremented, the entry it currently points to is decremented by one. The victim entry address (which is nothing but the forward pointer value to the data-victim) is inserted in a hash map table with the request address as the key. The location of the invalid tag store (i.e. the new reverse pointer, pointing to the tag store entry) is also saved in another hash map table, with address again as the key. The reverse pointer of the data-victim is used to invalidate the tag store entry that the victim is currently mapped to. Then the data is evicted.

In case when there is no invalid entry in the set, LRU replacement policy is used to select the tag victim and evict the data store entry pointed to by the tag store using the forward pointer. To maintain integrity and avoid explicit checks, we insert the tag and data store entry address in their respective hash maps as described above. Once the new data is allocated, the pointers are set appropriately in the tag and data store entries to point to each other by fetching them from the hash tables using address as key and then the hash map entries are deleted. The V-Way cache replacement was also tried on L1 cache, with LRU replacement in L2 cache. The results obtained from this experimental setup are also provided.

B. SLICC Implementation

Initially we explored the Ruby Memory Model by implementing local reuse replacement for L2 cache with the default tag-data store structure. The next step was to decouple the tag and data stores, thus eliminating the static mapping between the two. Our Ruby model based implementation used MESI coherence protocol with additional intermediate transition states to ensure data consistency. Each transition in the model is triggered based on the events controlled by specific SLICC implementation.

In case of L2 cache entry miss, the controller tries to find the entry in the cache, depending on the current state (MESI or intermediate state) we check whether there exists a valid tag store entry or not. In case a valid tag store entry exists, which means the write back has been issued but yet not acknowledged. we stall the pipeline to wait for the write back stage to complete before proceeding further. If there does not exists a mapping in the tag store for the current L2 cache entry, we traverse the corresponding tag store set to find an invalid tag store way. In case an invalid tag entry is found, we use global reuse replacement policy to find the victim data store entry, followed by an update to tag and data store structure. If no invalid tag store entry is found, we use LRU replacement policy to invalidate the least recently used tag store entry. We obtain the corresponding data store entry to evict using the forward pointer of the tag store entry. Once the SLICC implementation inquires for the entry to evict, it receives the pointer to the data store entry and triggers L2 replacement for that data entry.

During deallocation of the data store entry, we need to invalidate its old tag store entry so that it can be reused. Thus while allocating the data entry we map the recently invalidated tag store entry to currently allocating data entry. The implementation also updates the reverse pointer for the allocated data store entry and forward pointer, valid bit and address tag for the mapped tag store entry. After allocating the data store entry, a lookup is performed using the tag store structure to set the corresponding MRU bit. The look up compares the address tag of each tag store entry of the corresponding set to get the tag store entry match. Once matched, the forward pointer gives the index to the data store entry to set. While implementing the abstraction of tag store over data store, we ensured that the SLICC interface dependency on the data store remains intact thus avoiding the possibilities to modify the default MESI state transition flow. Our implementation also ensures the compatibility across the difference levels of cache hierarchy such that the default L1 cache implementation with LRU replacement policy works with along with the proposed modified L2 cache implementation.

C. Cache Configuration

We have used the following parameters for our baseline and V-Way experiments:

Parameter	Value
L1 I-Cache	16kB; 64B line-size; 2-way with LRU replacement
L1 D-Cache	16kB; 64B line-size; 2-way with LRU replacement
Baseline L2	256kB; 128B line-size; 8-way with LRU replacement
ROB entries	128
Memory Access Latency	300 ns

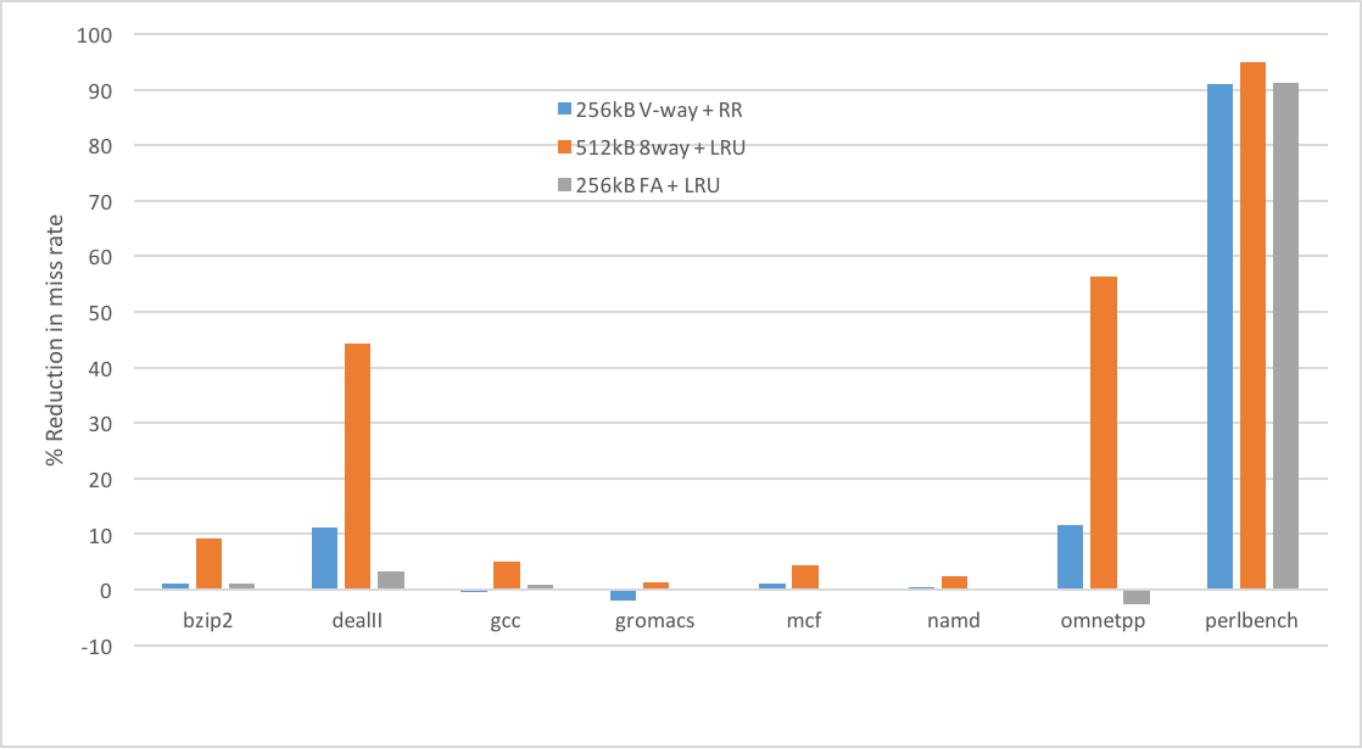

Figure 2. Reduction in miss rate with : V-way cache (TDR=2), double sized baseline and fully-associative cache

IV. Results/Observations

The percentage reduction in miss rate for lbm and sjeng benchmarks were very small and have been excluded from the graph. These are provided in the table below:

Benchmark	V-Way	Double Sized Baseline	FA + LRU
sjeng	0.175889	0.080075	0.378211
lbm	0.007086	0.006149	0.100707

V. Analysis

Performance results have been obtained for benchmarks from the SPEC CPU2006 suite which have been compiled for the Alpha ISA. The benchmarks were forwarded to a sim-point value with reasonably high weight and a 500 million input set was simulated for the classical cache based implementation with a warmup of 10 million instructions.

When using the Ruby Memory Model, fast forwarding requires use of MOESI hammer protocol with Out-of-Order CPU instead of atomic CPU. Due to time constraints, the benchmarks were not forwarded for the Ruby Memory Model based implementation and the first 500 million instructions were simulated. The implementation by Qureshi et al. [1] utilizes the SPEC 2000 benchmark suite. There exist only 4 related benchmarks in the SPEC2006 suite which are perlbench (successor of perlbmk), bzip2, gcc and mcf. The related benchmarks in the SPEC2006 suite are later versions of those in the SPEC2000 suite. It must also be noted that the peak allowed physical memory consumption of SPEC2006 benchmarks is 1GB as against 200MB for the SPEC2000

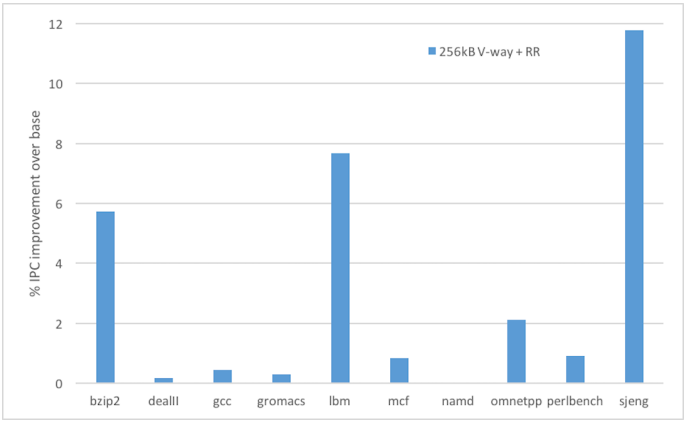

Figure 3. V-Way IPC improvement over baseline

benchmarks. All the benchmarks have been run with the *ref* input sets. The *test* input set was used for bzip2 since the ref input set caused the simulation to exit early (due to exit() call by the benchmark). Along with comparison between v-way and baseline, we have also simulated the system for 256kB fully associative L2 cache with LRU and 512kB L2 cache. These runs help us evaluate whether the working set is larger than the cache set (in which case we need greater number of tag stores with global replacement) or we are in need of more data stores along with more tag stores. The expected behavior of the benchmarks was obtained from documentations [3], [4], [5]. The following analysis is based on a sample set from the data collected. Extensive amount of data was collected to visualize the cache size variations, tag to data ratio variations

and their effect on IPC (These data points are available on request).

A. L2 V-Way analysis

1) 400.Perlbench: In case of perlbench, memory operations are 27% of entire workload. However, L2 miss rate is about 2% due to repeated operations on a group of same data sets. Global replacement policy enables replacement of data victims from less frequently used sets giving a much lower miss rate than baseline. Improvement in IPC is lower than other benchmarks despite very high miss rate reduction because memory accesses constitute a small part of the entire benchmark. Also, the number of I-cache misses are high (compared to other benchmarks) which results in a pipeline stall. The pipeline stalls for more time due to increased L2 hit latency, which causes lower IPC improvement compared to that expected from reduction in percentage misses.

2) 401.bzip2: In case of bzip2 we observe that the total number of memory operations is 35% of entire workload and the L2 miss rate is high. However, we see that fully associative LRU does not provide any significant improvement over baseline which shows that the working set is much larger than the cache size and continuous replacements are causing a lower miss rate change. Doubling the cache size shows significant improvement in performance which leads us to conclude that doubling tag stores will not provide any significant improvement in number of misses. However, we observe a significant improvement in IPC because the reduced misses form a large part of the entire code.

3) 435.gromacs/444.namd/403.gcc: In case of gromacs (and namd) we observe that the total number of memory operations is 36% (and 29%, 30%) of all operations. Looking at the FALRU and cache size doubling cases, we observe that similar to bzip2 these benchmarks require more data stores along with tag stores. In these cases, the V-way cache gives more misses but the IPC increases by about 0.3% (and 0.1%, 0.44%). This increase is mainly due to better victim selection by the global reuse replacement compared to LRU policy.

4) 447.dealII: In case of dealII, we observed that the working set is larger as compared to data store entries. The number of tag-store entries required to increase the number of hits is dependent largely on the replacement policies. Global RR performs well in this case which selects better victims and allows more frequently accessed cache lines to remain for longer time, resulting in reduced miss rate if certain cache lines are being accessed neither too distant nor too immediate. In case of double sized baseline, the increase in number of data store and tag store entries result in reduction in miss rate as compared to V-way implementation.

5) 471.omnetpp/429.mcf: In case of above benchmarks, the change in misses is due to reasons similar to dealII. However, the total number of L2 misses and memory accesses is higher for thee benchmarks compared to dealII which reflects in correspondingly high IPC improvement.

6) 458.sjeng/470.lbm: Baseline sjeng implementation exhibits 10% L2 miss rate. In case of sjeng, we observe that fully associative cache provides significant improvement over

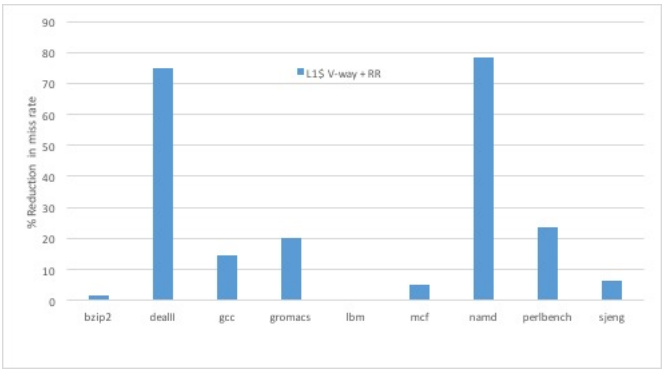

Figure 4. L1 Cache V-way percent miss rate reduction

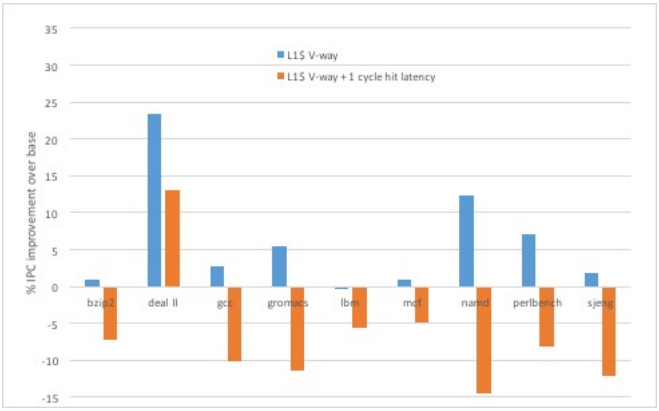

Figure 5. L1 IPC change V-way and V-way with increased hit latency

baseline which show that multiple accesses to the same set are being performed. Total change in the miss rate is about 0.2% but the IPC changes by about 11%. This is due to better victim selection by the Reuse Replacement policy.

B. L1 V-Way analysis

As seen from the graphs given below, the number of misses have reduced drastically when we implement V-way cache with RR policy on L1. However, since the increase in hit latency for the L1 cache caused by this structure was not considered in the first case and hence, the IPC also improves accordingly. When we consider the latency overhead caused by V-way and RR, the IPC reduces as shown in the second graph. For first level cache, hit latency being the more important design consideration the V-way implementation cannot be used on L1 cache even though we have exceptional miss rate reduction.

VI. Caveats

In order to overcome invalid transitions in Ruby model implementation, we had to stall the pipeline which led to minor reduction in performance. Extensive knowledge of the SLICC language and coherence protocol implementation would have allowed us to explore the SLICC interface better and obtain an even better representation of the actual architectural implementation.

VII. Conclusion

A total of 15 benchmarks were run for varied instruction counts to capture the correct behavior of our V-way implementation with the global replacement policy. A 256kB, 8-way second level V-Way cache using Reuse Replacement has given an average reduction of 11.41%(compared to 13% in [1]) in miss rate and an average IPC improvement of 3%(compared to 8% in [1]) over the baseline. We also observe that the Reuse Replacement policy is completely implementable for both L1 and L2 caches and performs better by selecting a better victim leading to improvement in IPC. Thus, varying the associativity on a per-set basis by increasing the number of tag-store entries provides a better alternative over fully associative caches and standard set-associative caches.

References

[1] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way Cache: Demand Based Associativity via Global Replacement. *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *In IBM Systems journal*, pages 78–101, 1966

[3] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006

[4] J. L. Henning, "SPEC CPU2000 memory footprint," online at *http://www.spec.org/ cpu2000/analysis/memor*y

[5] J. L. Henning, "Performance Counters and Development of SPEC CPU2006," *Computer Architecture News*, vol. 35, no. 1, 2007