Comparison of Several L2 Cache Schemes

Zubin Kane : V-Way Cache Akash Shankar: Adaptive Insertion Policy

https://github.com/PurdueECE565f21/pa1-Sclamy (Zubin's Repository)

Abstract

Set Associative caches suffer from two types of misses: capacity miss and conflict miss. While capacity misses can be reduced by simply increasing cache size, conflict misses are typically a result of low associativity, which is required by cost. Many schemes have been proposed to reduce conflict misses on 2-8 way set associative caches. We evaluate two novel schemes (Dynamic Insertion Policy, V-Way Associative Cache) and compare their results.

1. Introduction

Fully Associative Caches are capable of minimizing conflict misses, given the right replacement algorithm. Since the entire cache is effectively one set, the replacement algorithm responsible for selecting insertion victims is global, and is therefore capable of fully utilizing cache capacity as needed, and as the algorithm dictates.

However, these caches are often too costly to create and run, and Set Associative Caches are used instead. The lower associativity of set associative caches greatly simplifies cost and replacement algorithm complexity, since only a handful of potential victims need be considered upon each insertion. Unfortunately, lower associativity also leads to greater conflict misses - for instance, a single set may be thrashed by multiple sources, while another set remains empty. It is necessary that we seek to minimize conflict misses, given this practical constraint of low n-way set associative caches.

2. Motivation

2.1. Strict Insertion Policy Approach

The LRU insertion policy is the most commonly used replacement policy. For workloads that have a working set lesser than the cache size, LRU succeeds in maintaining good performance. But, for memory-intensive workloads that have a working size greater than cache size, LRU is susceptible to thrashing, and does not yield good performance.

The Dynamic Insertion Policy (DIP) solves this problem. It can dynamically choose between LIP (LRU Insertion Policy), which places the incoming lines in the LRU position instead of MRU position, and the BIP (Bimodal Insertion Policy), which follows LIP, but for a small percentage of cases it places the incoming lines in the MRU position. The proposed insertion policy does not require any changes to the existing cache structure, are trivial to implement, and have a storage requirement of less than two bytes. Since a miss is L2 produces a long stall, we are interested in managing the L2 cache efficiently.

2.2. Cache Structure Approach

Many conflict misses are the result of the strict set structure of set-associative caches. In every conflict miss, we are forced to choose a victim from a limited subset of options, when there exists a better victim outside the subset. From a different perspective, the set we are inserting into would benefit from *more* capacity, while the set containing the "ideal" victim could handle *less* capacity. In most cases, sets in set-associative caches have unique demands for space, with some sets demanding more than others, yet the traditional structure of set-associative caches treats all sets equally.

The Variable-Way Set Associative cache increases general associativity by dynamically changing cache structure, providing larger capacity to sets with higher demand. Since the overall capacity of the cache is constant, this set capacity is taken from sets with lower demand, via a global replacement algorithm. To maintain minimal replacement cost, set capacity is upper-bounded by an integer multiple of starting capacity, and while any set is at this upper bound, it performs local replacement. Global replacement is therefore sparsely used. In addition, the global replacement algorithm is lightweight, since it does not need to choose the best

victim, only a decent victim. Overall, this change to set-associative cache structure incurs very little additional cost, and is capable of significantly increasing general associativity, effecting reduced conflict misses.

3. Dynamic Insertion Policy

In this project, DIP (Dynamic Insertion Policy) is implemented using a technique called set-dueling. The key insight in set dueling is that cache behavior can be approximated with a high probability using a few sets in the cache. Set Dueling dedicates a few sets of cache to both the competing policies (BIP and LIP). The policy that incurs fewer misses on the dedicated sets is used for the following remaining sets. This implementation is called DIP-SD.

The N be the number of sets in cache and let K be the number of sets dedicated to each policy. Now the cache is logically divided into N/K sets. Each region is called a constituency. One set is dedicated from each constituency to each competing policy. Two bits associated within each set can then identify whether the set is a follower set or a dedicated set. A policy for selecting dedicated sets: for a cache with N sets, the set index consists of log2(N) bits out of which the most significant log2(K) bits identify the constituency and the remaining log2(N/K) bits identify the offset from the first set in the constituency. The complement-select policy dedicates to LRU all the sets for which the constituency identifying bits are equal to the offset bits. Similarly, it dedicates to BIP all the sets for which the complement of the offset equals the constituency identifying bits. Thus for the baseline cache with 1024 sets, if 32 sets are to be dedicated to both LRU and BIP, then complement-select dedicates set0 and every 33rd set to LRU, and Set 31 and every 31st set to BIP.The sets dedicated to LRU can be identified using a five bit comparator for the bits [4:0] to bits [9:5] of the set index. Similarly, the sets dedicated to BIP can be identified using another five bit comparator that compares the complement of bits [4:0] of the set index to bits [9:5] of the set index. A 10-bit PSEL counter is used. A miss incurred in the sets dedicated to LRU increments the counter, whereas a miss incurred in sets dedicated to BIP decrements the counter. If MSB of PSEL is 0, the follower sets use LRU policy, otherwise the follower sets use BIP.

4. Variable-Way Associative Cache

The V-Way Cache implementation requires three key components:

- Set structure capable of emulating variable-size sets
- Lightweight global replacement policy
- Strong local replacement policy

In a typical set-associative cache, memory contains two types of entries: *tag-store* entries and *data-store* entries. Each data-store entry is simply the memory containing a block within a set, and each tag-store entry is a bit-mask “tag” for the distinct memory address stored within a linked data-store entry. To perform lookup, the set is determined from “index” bits of an address, then all entries within that set are searched for the “tag” bits. Each tag-store entry is linked to a single data-store entry, one-to-one.

To produce variable-size sets, we critically separate tag-store entries from data-store entries. Our cache has limited block storage capacity, so the number of data-store entries remains unchanged. We create N*X tag-store entries, where N is the number of data-store entries and X is the “maximum set size” integer multiple of our base set size. We then group these tag-store entries into sets of the maximum set size. This effectively creates X times the number of set, with the same number of entries per set as before, except that on average, only 1/X of the tag store entries in any given set will be in use.

Note that we consider X to be a multiple on maximum set size given that since the new average use of each set is (set_size)/X, the maximum use in any set is X*(set_size)/X === (set_size). While we can change X to create highly variable associativity, this theoretically has highly diminishing results, since a large read or write to a single set has the potential to upset much more of the cache outside that set. For the duration of this paper, we will use X=2.

The separate tag-store and data-store entries can now be connected by the following scheme. If an insertion is performed into a set with empty tag-store entries, and there are unused data-store entries (during warmup, presumably), then the unused data-store entry is assigned to the unused tag-store entry. If an insertion is performed into a set with empty tag-store entries, but there are no data-store entries, then lightweight global replacement is used to find a victim somewhere in the cache. That victim’s tag-store entry is invalidated, the new tag-store entry is validated, and the data-store entry used by the

victim is given to the new tag-store entry. If an insertion is performed on a full set, regardless of the number of unused data-store entries, then local replacement is used to choose a victim from the set, and that victim's tag-store entry is changed to reflect the insertion. Throughout this scheme, each tag-store entry contains several attributes, such as a dirty bit, several last-used bits, and two saturating reuse bits which will be discussed presently. Writebacks are performed whenever evicting a dirty entry. Note that unlike a traditional set-associative cache, tag-store entries will be validated and invalidated even after warmup, and never will all tag-store entries be valid.

The lightweight global replacement policy proposed alongside the V-Way Cache by Qureshi, Thompson, Patt is referred to as Reuse Replacement. This scheme maintains a two-bit saturating counter on each used data-store entry, which is incremented upon use. When global replacement is needed, a single pointer is used to loop through data-store entries. If an entry is greater than zero, it decrements it and moves on. If an entry is zero, it returns it and moves on once more. The next use will return to the entry it stopped on. In this way, this global replacement algorithm can often find a minimally-used victim without traversing the majority of data-store entries, unlike most global-replacement policies.

The local replacement policy used is LRU (Least Recently Used). Whenever a tag-store entry is used, its last-use is set to zero, and every other tag-store *in the same set* is incremented. This algorithm is common, simple, and does a fair job of maintaining often-recurring tags within a set.

5. Experimental Methodology

Since the objective of these schemes is to reduce conflict misses, we evaluate performance on the metric of miss rate. Specifically, we compare "Percentage reduction in miss rate" of a given scheme over some common baseline. All schemes are implemented using the Gem5 simulator, on an Out of Order CPU.

5.1. Cache Configurations

The various cache configurations used are shown in Table 1. The L1D (L1 Data) and L1I (L1 Instruction) cache configurations were unchanged across tests, while a single configuration of the L2 cache was selected for each test. Cache sizes chosen to maximize conflict miss impact for comparison. The baseline configuration used a traditional 8-way L2 cache with LRU replacement.

Table 1. Cache Configurations

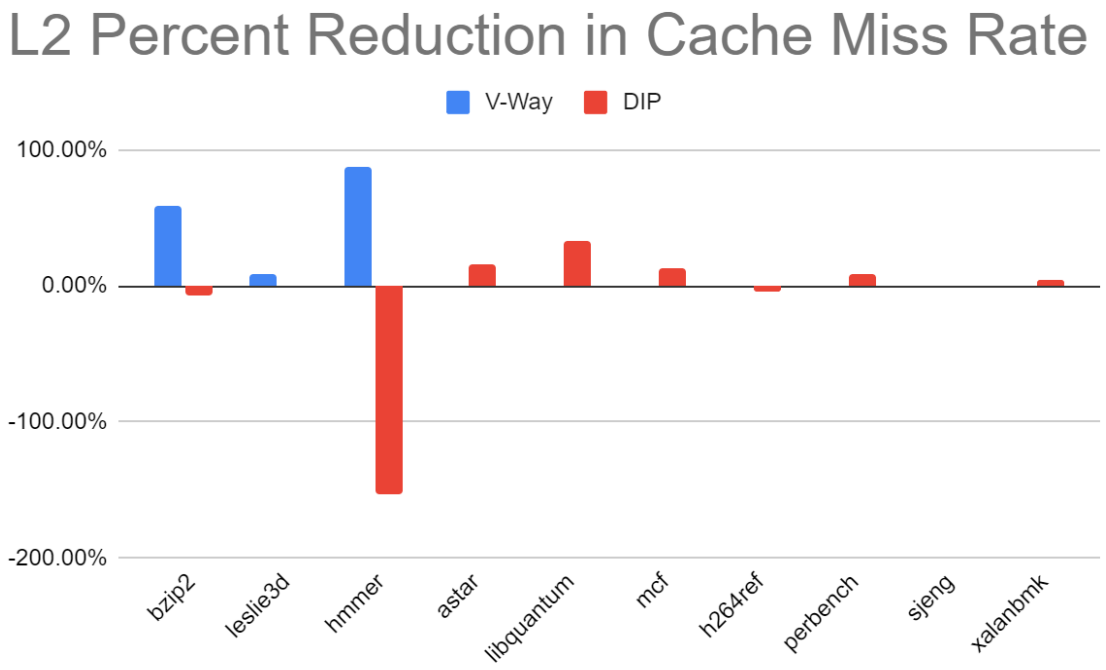
L1 I_Cache	4kB; 2-way
L1 D-Cache	4kB; 2-way
Baseline L2	16kB; 8-way
L2	16kB; 8-way with DIP repl.
V-Way L2	16kB; 8*2-way with Reuse/LRU

5.2. Procedure

Each cache configuration was tested with several benchmarks from SPEC2006.. Warmup was performed for 10 Million instructions, followed by testing for an additional 50 Million instructions. Additionally, testing was performed on various other cache sizes and instruction counts, with significantly higher instruction counts and lower cache sizes often demonstrating marginally lower miss rates.

6. Results

(note: V-Way results below only shown for first 3 benchmarks)



6.1. DIP Discussion

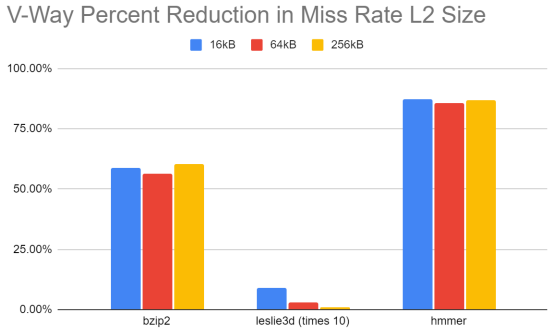
One of the major reasons for not achieving the expected cache performance from the DIP policy is that the working set of the benchmarks that were run were not big enough for the PSEL counter to select the BIP insertion policy. It turns out that the benchmarks mcf, libquantum, and perlbench incur the highest miss rate of all them all, and therefore stand a better chance of exploiting the DIP policy. For certain benchmarks like hmmer, bzip, the access pattern is such that the PSEL counter remains close to neutral (zero), and therefore go on to incur a negative reduction in miss rate.

6.2. V-Way Discussion

Overall, the V-Way cache implementation exceeded expectations with regards to the initial paper implementation. While the paper did show results for different benchmarks vary from -10% reduction up to 95% reduction, the authors only managed to achieve a 2% reduction on bzip2. We theorize that this is the result of the number of instructions tested, since running bzip2 on 50 million and 100 million instructions indicated a small but slow decline in miss rate reduction. Since the authors ran for 15 billion instructions, it is likely that the effectiveness of our implemented cache would decrease to similar levels if run for that long.

We also observed changes resulting from increasing cache size (Table 2), which in some cases varied only slightly, although large cache greatly diminished the returns from leslie3d.

Table 2. Varying L2 Cache Size



7. Conclusion

It is clear from our results that the V-Way cache in general produces a better performance than DIP, which relies on the particular benchmark and the specific size of the caches. We observe that V-Way seems to excel in benchmarks where DIP performs poorly. This seems to indicate that these techniques are useful in disparate situations, and that if one appears to give poor performance for a certain task, the other may perform quite well. Both are quite novel approaches to reducing overall conflict miss rate.