Advanced Computer Architecture

CS622 Assignment 1: Report

Debanjan Chatterjee (20111016) and PJ Leo Evenss (20111038)

Group: 13

1. Introduction

In this assignment we have simulated a two-level cache hierarchy, and passed the L1 cache miss trace of six SPEC CPU 2006 applications: bzip2, gcc, gromacs, h264ref, hmmer, sphinx3. The two-level cache hierarchy are L2 and L3 caches with the following ABC parameters. The L2 cache size is 512 KB, with block size 64 bytes, and 8-way set associative, whereas the L3 cache size is 2 MB, with block size of 64 bytes and 16 way-set associative.

There are two objectives: Firstly, to report the number of L2 and L3 cache misses for the six applications when they implement three separate polices: inclusive, exclusive and NINE. Secondly, for the inclusive policy, the L3 cache misses need to be categorized into the type of cache miss, i.e. cold, capacity, and conflict misses. So far, the caches have been made to implement the Least Recently Used (LRU) replacement policy. An implementation of the L3 cache with Belady's optimal replacement algorithm for the fully associative case, to report the number of cold and capacity misses, has also been carried out.

2. Result Analysis

L2 and L3 cache misses for inclusive, NINE, and exclusive policies.

The results for L2 cache misses and L3 cache misses are displayed in Table 1 and Table 2 respectively. The first observation to be made is that for any selection of trace file and policy, the number of L3 cache misses is significantly fewer than the respective L2 cache misses, which would mean that there were a considerable number of L3 cache hits, which can be easily found out by the following equation.

$$L3\ cache\ hits = L2\ cache\ misses - L3\ cache\ misses$$

Now let us look into the variation in L2 and L3 misses across the three policies:

2.1.1. L2 misses for inclusive policy is greater than NINE and exclusive.

To explain this observation, we need to take a closer look at what happens in case of Inclusive policy, when a L3 cache block is evicted. Since the cache hierarchy needs to maintain the inclusion property: L2 ⊆ L3, the L3 cache victim block, on eviction, also invalidates the corresponding block from the L2 cache (if present).

To make matters worse, when there is a L2 cache hit, the LRU order is only updated for the L2 cache set, while the LRU order remains unchanged for L3 cache. Thus, a cache block enjoying a lot of L2 hits might end up being the least recently used cache block in L3 cache, and end up being evicted when there is a need for replacement, hence simultaneously invalidating the corresponding block from L2 cache, in spite of it being accessed recently. NINE and exclusive policies does not suffer from this drawback as there is no strict requirement to maintain inclusivity, hence they suffer fewer L2 cache misses than inclusive.

2.1.1. L2 misses are equal for NINE and exclusive policies.

This observation can be explained, if we can intuitively show that the contents of L2 cache remains same for both NINE and Exclusive policies. Let us look at the various cases, focusing on the L2 cache contents. Initially, the L2 cache is empty for both NINE and excusive policy.

- In case of L2 cache hit, LRU order is updated for both exclusive and NINE L2 cache sets, in the same way.
- L3 hit after missing in L2: for both NINE and exclusive, the block is transferred to L2 cache.
- L3 cache miss, after missing in L2: the block is brought to L2 in both NINE and exclusive policies. If there is a L3 cache block eviction, the corresponding block need not be invalidated in the L2 cache, for both the policies.

Therefore, we can conclude that the contents for the L2 cache remains same for exclusive and NINE, throughout the execution, which will lead to an equal number of L2 misses for both the policies.

Table 1: L2 cache misses for inclusive, NINE, exclusive policy.

Trace files	**Inclusive**	**NINE**	**Exclusive**
bzip2	5398166	5397576	5397576
gcc	3036461	3029809	3029809
gromacs	336851	336724	336724
h264ref	969678	965624	965624
hmmer	1743421	1735322	1735322
sphnix3	8820349	8815130	8815130

2.1.3. L3 misses in inclusive policy > NINE policy > exclusive policy

The capacity of a two-level cache hierarchy with exclusive policy will be maximum compared to the same cache hierarchy exercising NINE or inclusive policies. The reason can be attributed to its strict maintaining of the property, L1 ∩ L2 =∅, thus granting it the largest effective capacity. Therefore, the number of L3 misses will be least for the exclusive case.

Now, in order to explain the difference in L3 misses for inclusive and NINE, we need to consider the course of action when a L3 cache block is evicted. For inclusive, the corresponding cache block needs to be invalidated in the L2 cache, but for exclusive no invalidation is necessary. Therefore, that cache block continues to persist in the L2 cache block and enjoy future L2 cache hits, therefore reducing the number of L2 misses for NINE, as compared to inclusive

Table 2: L3 cache misses for inclusive, NINE, exclusive policy.

Trace files	Inclusive	NINE	Exclusive
bzip2	1446388	1445846	889221
gcc	1373402	1366248	1242824
gromacs	170531	170459	159302
h264ref	342146	333583	143681
hmmer	391226	376344	300046
sphnix3	8207362	8205144	7220776

2.2. Classifying cache misses into cold, capacity and conflict misses

2.2.1. L3 miss classification for inclusive policy executing LRU replacement

In the previous section, the L3 cache misses for the Inclusive case (LRU replacement) have been classified into cold, capacity and conflict miss and the results are shown in Table 3.

The cold misses can be calculated by finding out the number of unique cache blocks accessed in the application's address space. For calculating the capacity misses, we simulate the L3 cache keeping the same cache size and block size, but making it fully associative, i.e. reduce the number of conflict misses to zero. Now, the total number of L3 cache misses is recomputed, and the number of capacity misses can be calculated from the following equation.

$$L3\ capacity\ misses = L3\ cache\ misses\ (fully\ associative) - L3\ cold\ misses$$

The number of conflict misses in L3 cache (set associative configuration from the previous section) can be calculated by:

$$L3\ conflict\ misses = L3\ misses\ (set\ associative) - L3\ cold - L3\ cpacity\ misses$$

Table 3: L3 cache misses (exercising inclusive policy and implementing LRU replacement) classified into cold, capacity and conflict misses

Trace files	**Cold**	**Capacity**	**Conflict**
bzip2	119753	1241648	84987
gcc	773053	596871	3478
gromacs	107962	61406	1163
h264ref	63703	272177	6266
hmmer	75884	301140	14202
sphnix3	122069	8265179	-179886

From table 3, we observe an anomaly, that is the number of L3 conflict misses for 'sphnix3' is negative, this would mean that the total number of L3 misses (cold and capacity) for fully associative implementation is greater than the number of L3 misses for set /associative implementation. To explain this occurrence, we need to take a deeper look at both the implementation and see where they might differ. Since we are simulating a two-level cache hierarchy with the L2 cache remaining constant (set associative LRU), and the L3 cache executing LRU, but differing in associativity, hence there will be difference when it comes to the LRU L3 cache victim selection in the set associative and the fully associative case. For fully associative implementation, the LRU block in the L3 cache needs to be evicted, whereas for set associative, the LRU block of the target cache set needs to be evicted. Furthermore, the inclusive policy of the cache hierarchy mandates that the L3 cache block when evicted, needs to invalidate the corresponding block from the L2 cache. Also, on a L2 cache hit, the LRU order of the L2 cache is updated while the LRU order for L3 remains unchanged. Thus, a cache block enjoying a lot of L2 hits might end up being: the least recently used cache block in the entire L3 cache for fully associative implementation, or the least recently used cache block in that particular cache set for set associative implementation. Now, if there is a need for replacement in L3, the cache block which received a lot of L2 hits, will certainly be evicted for fully associative thus invalidating it from L2 as well, but might not be evicted for the set associative case, if the newly loaded cache block does not belong to the same cache set, hence it will continue to enjoy a few more L2 hits. Therefore, possibly increasing the number of L3 cache misses for fully associative implementation. This drawback of inclusion policy, is

corrected when Belady's optimal policy is used instead of LRU, as we can observe from the results in Table 3 and Table 4, the number of L3 capacity misses for 'sphnix3' is 8.26 million for LRU, which falls down to 2.9 million for Belady's optimal policy, as it picks the L3 cache victim based on future access patterns.

2.2.2. L3 miss classification for inclusive policy executing Belady's optimal replacement

Now, for the above fully associative implementation, we simulate the L3 cache to execute the Belady's optimal policy (results in Table 4) and compare the number of L3 misses in both the replacement policies. Conflict misses are zero in case of fully associative implementation.

Table 4: L3 cache misses (exercising inclusive policy and implementing Belady's optimal replacement) classified into cold and capacity.

Trace files	**Cold**	**Capacity**
bzip2	119753	417083
gcc	773053	166236
gromacs	107962	35292
h264ref	63703	47902
hmmer	75884	77563
sphnix3	122069	2946511

For the results, displayed in Table 3 and Table 4 we observe that,

- The number of cold misses remain unchanged in both LRU and Belady's optimal replacement policies. The reason is simple, as cold misses is independent of the replacement policy used, and it is strictly equal to the number of unique cache blocks accessed by the application's address space
- In case of capacity misses for each trace file, the number is much lower in case of Belady's optimal replacement policy than LRU, since optimal replacement algorithm is theoretically the most efficient replacement algorithm as it can look into the future memory access pattern, and replace the cache block, whose next access is farthest in the future.