CS622 Project
A Study of Dynamic Associativity Management Using Fellow Sets in LLC

Alan Nair (Group 7)[a]

[a]*Submitted to Prof. Mainak Chaudhary*

1. Introduction

The project aims to implement and analyze the performance of a method of cache resizing called **CMP-SVR** [1]. In this technique, the sets in LLC are partitioned into fellow groups. If a set is being heavily used, it can use the reserve storage provided by other sets in its fellow group. The assumption here is that heavily used cache sets are evenly distributed across fellow groups.

2. Approach

First we divide the sets in LLC into fellow sets of some size k. The sets are assigned to fellow sets serially.

$$fellow\_group\_number = set/k$$

Now, accessing a block in LLC involves the following steps:

- Given address, find its set. Check if the required block is present in that set. If yes return the way ID. Update LRU states of the blocks in the current set accordingly. Record an LLC hit.
- If not found, then find the fellow group number of the set, and hence all sets in that fellow group. Check if the required block is present in any set inside this fellow group. If yes, swap the block with the LRU block of the current set (leaving their LRU states intact). Return the way ID of the block where it now resides (the one that originally belonged to the block in the current set that Update LRU states of the blocks in the current set, as if it was a hit in the first step. Record an LLC hit.
- If the required block has still not been found, record an LLC Miss.

3. Implementation

This paradigm is tested by implementing it on a ChampSIM simulator [2]. Edits were made in the following files within the source code.

- **src/cache.h**: We add two members into the CACHE class, denoting the number of sets in fellow groups and fellow group size and a method to calculate fellow set ID.
- **src/cache.cc**: Definition for the method to calculate fellow set ID and changes in the functions check_hit and invalidate_entry.
- **src/base_replacement.cc**: Changes in the functions lru_victim and lru_replacement.

A cache executable (using the original source code) is generated with one core, with a bimodal branch predictor and LRU replacement policy (and default prefetchers in ChampSim). This will be our baseline cache. Another cache executable (using the aforemetioned edits to the source code) is generated with the same options. This will be the cache for the paradigm we are about to test.
We shall run a benchmark on both. The benchmarks, 600.perlbench_s-210B, 602.gcc_s-734B and 603.bwaves_s-3699B. is taken from SPEC 2017 and were used in DPC-3. 50 million instructions were used for warmup. Simulation was done for 200 million instructions.

References

[1] H. K. K. Shirshendu Das, Dynamic associativity management using fellow sets, IEEE.
[2] Champsim simulator, https://github.com/ChampSim/ChampSim.

4. Experimental Evaluation

We expect the outcome to be such that the misses in LLC would reduce (under the assumption that heavily used sets are evenly distributed across the cache.

Evaluation on PERLBENCH

	Baseline Cache			CMP-SVR Cache		
Nature of Access	Hits	Misses	Total	Hits	Misses	Total
LOAD	216	8838	9054	205	8849	9054
RFO	1	972	973	1	972	973
WRITEBACK	914	0	914	912	2	914
TOTAL	1131	9810	10,941	1118	9823	10,941

Evaluation on GCC

	Baseline LLC			CMP-SVR LLC		
Nature of access	Hits	Misses	Total	Hits	Misses	Total
LOAD	15847	1869769	1885616	10334	1875282	1885616
RFO	29	2923	2952	23	2929	2952
WRITEBACK	17502	3615	21117	4170	16947	21117
TOTAL	33378	1876307	1909685	27304	1882381	1909685

Evaluation on BWAVES

	Baseline LLC			CMP-SVR LLC		
Nature of access	Hits	Misses	Total	Hits	Misses	Total
LOAD	11	3506	3517	1	3516	3517
RFO	0	17532	17532	0	17532	17532
WRITEBACK	17556	0	17556	17556	0	17556
TOTAL	17567	21038	38605	17557	21048	38605

5. Conclusion (Possible Explanation)

It is seen that for the perlbench benchmark, there was hardly any difference at all between the two caches, for load and rfo accesses, but fared slightly worse for writeback accesses. This is somewhat inconsistent with our expectation contradicts the findings in the original paper[1]. One reason for this could be that heavily used sets are unevenly spread out across all fellow groups. So, if multiple heavily used sets are in the same fellow group, then any gains by this paradigm can get nullified by the possibility that one heavily used set might end up deleting another heavily used set's blocks from LLC.