ECE 565 Course Project: ZCache

Abhishek Damle
Purdue University
West Lafayette, USA
adamle@purdue.edu

Sujay Pandit
Purdue University
West Lafayette, USA
pandit8@purdue.edu

Abstract

ZCache [1] aims to reduce conflict misses by improving associativity without increasing the number of physical cache ways. We implement ZCache using GEM5 and compare its performance to set, and skew associative caches for various numbers of cache ways, cache sizes, and SPEC 2017 benchmarks. We identify configurations where conflict misses constitute a significant portion of cache misses and find that ZCache reduces the miss rate by 27.06% compared to set associative caches and 1.98% compared to skewed associative caches for these configurations. Among these configurations, ZCache also improves IPC by 15.58% compared to set associative caches and 1.17% compared to skewed associative caches.

I. Introduction

In modern computer architectures, the main memory is orders of magnitude slower than the processor. Caches alleviate this bottleneck by providing hierarchical storage layers between the processor and main memory. This hierarchy is an integral part of computer architecture and consists of memory devices that trade-off between speed and size to realize improved overall system performance.

Cache misses play a critical role in cache and overall system performance. A cache miss occurs when the processor attempts to access data that is not located in a level of cache, causing the cache to fetch the data from higher-level memory such as another cache level or RAM. Therefore, cache misses can severely degrade system performance since memory accesses are orders of magnitude slower than the processor.

Cache misses can be classified into three categories: compulsory misses, capacity misses, and conflict misses. Conflict misses occur in set associative caches when multiple blocks that map to the same set are accessed in rapid succession. If the number of such blocks exceeds the number

of ways in the cache, the cache blocks must be moved into and out of the cache, increasing the miss rate and lowering the performance of the cache.

Increasing the associativity by increasing the physical ways of a cache is a naive method to reduce cache conflict misses. Increasing the number of physical ways increases the associativity of the cache and thus reduces the number of conflict misses. However, the latency of cache hits is also increased. Therefore, increasing the number of physical cache ways is not a suitable solution for improving the overall performance of caches. ZCache aims to improve the efficiency of cache associativity by decoupling associativity and the number of physical ways. The insight behind ZCache's design is that the number of replacement candidates determines associativity. Consequently, ZCache uses a series of hash functions to boost the number of replacement candidates, thereby increasing associativity without increasing the number of physical ways.

II. Background and Related Work

A. Skewed Associative Caches

Typical set associative caches use a single mapping function per cache way to map an address to the same set in each way. Therefore, address that conflict in one cache way also conflict in all other ways. On the other hand, skewed associative caches [2] use different hash functions per way to map an address to a set. Consequently, addresses that conflict in one way, do not have conflicts in the other ways. Spreading out conflicts in this manner reduces conflict misses of skewed associative caches compared to typical set associative caches [3].

B. Victim Caches

Victim caches are small, highly associative caches that temporarily store blocks evicted from the main cache [4]. Cache blocks leave the victim cache when they are either re-referenced or evicted. Since victim caches hold cache blocks recently evicted from the main cache, they allow evicted blocks to be re-referenced for a short period of time, thereby reducing conflict misses. However, victim caches increase the miss penalty and perform poorly for a large amount of conflict misses [1].

III. Implementation

This section describes ZCache functionality and our implementation using GEM5. We start with GEM5's skewed associative cache and modify it to enable ZCache functionality.

A. Cache Access

ZCache and skew associative caches follow the same procedure for cache accesses that result in hits/misses without replacement. Both caches determine a single block location per way using address bits and a unique hash function per way.

ZCache misses that result in replacements must be handled in two steps. These are the steps where ZCache differs from Skewed Associative caches. Before we dive deeper into the specific implementation, refer to table I that explains the minor additions to the GEM5 Cache Block structure.

TABLE I: GEM5 modification to Cache block structure

New fields	**Explanation**
bool replace_parent	This boolean field is set to True whenever a second-level candidate is selected for replacement. This field helps us later decide in the eviction logic that the parent must be moved in place of this block.
CacheBlk* parent	Pointer to the parent cache block. When the second-level replacement candidate is picked for replacement, the parent Cache block must be moved to its place. The new block will then move where parent block resided.

B. Replacement Candidates

In the first step, two levels of replacement candidates are generated. For an N-way cache, the N first-level candidates are generated by hashing the address of an incoming cache block with N hash functions. The addresses of these level 1 candidates are then each hashed with $N-1$ hash functions, resulting in $N(N-1)$ replacement candidates. In total, this process generates N^2 replacement candidates for an incoming block to a N-Way ZCache. Fig. 1. shows how nine replacement candidates are generated for a 3-Way ZCache using hash functions $H0, H1, H2$. In our implementation, the first-level candidates are called 'parents'. When the replacement logic chooses a second-level candidate for eviction, we set the 'replace_parent' as 'True' and set its parent pointer to point to the first-level Cache block whose address was used to find this block. For example, if the replacement logic chooses 'F' as the block to be evicted then we set its 'parent' pointer to point to 'B' cache block.

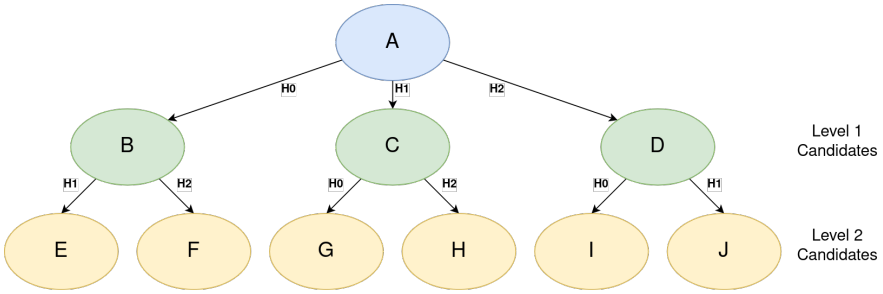


Fig. 1: Two levels of replacement candidates for a new cache entry, A.

C. Relocation

In the second step, the replacement candidate is evicted, and the incoming block is stored in the cache. We use a LRU replacement policy to select the best replacement candidate. Once the best replacement candidate is selected, we evict the candidate, relocate its parent candidate, and move the incoming block into the cache. Fig. 2. shows the relocation process when a level 2 candidate is evicted. We implement this logic in the eviction stage by first checking if the 'replace_parent' field is 'True' for the block to be evicted, if it is then after the block is evicted we use the 'parent' pointer to move the parent cache block to the location of the evicted block using GEM5's internal 'moveBlock' function. This takes care of moving all the metadata. However, the cache block structure also contains a 'data' pointer that contains the data of the block. We have to explicitly use 'std::memcpy' to copy over the data from the parent cache block to its new location. We then reset the 'replace_parent' field of the evicted cache block. Once the parent cache block has replaced the evicted block, we then place the new incoming block in the parent's location.

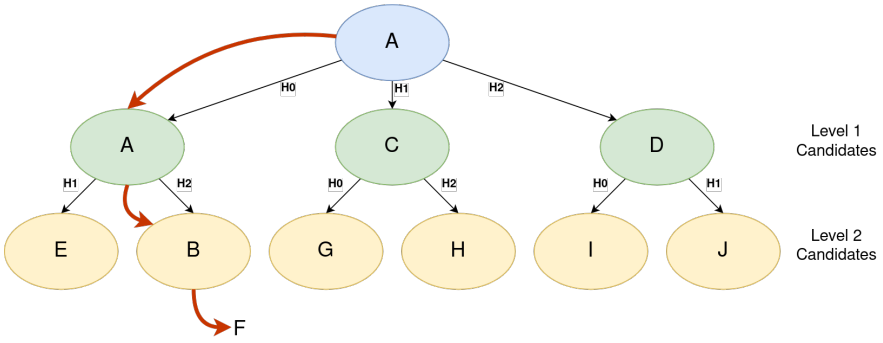


Fig. 2: Relocations when a cache block, F, is evicted and new cache entry, A, is inserted.

IV. Experimental Setup

We implement ZCache in GEM5 and compare its performance to skewed, set, and fully associative caches across a series of SPEC 2017 benchmarks. Namely, we measure cache performance using the cache miss rate and overall system performance using IPC. Table II shows the system configurations used for comparison. We limit our ZCache implementation to L2 cache. For our comparison results, we vary the cache type (set associative, skewed associative or fully associative), number of ways, and size of L2 caches to determine the effectiveness of ZCache.

We also use small L1 caches to let most of the requests reach and stress the L2 cache. This helps us better isolate and understand the impact of ZCache on system performance.

TABLE II: System Configurations

CPU	x86 timing simple CPU
L1 Caches	ways: 2 tag latency: 2 data latency: 2 response latency: 2 MSHRs: 4 targets per MSHR: 20 type: set associative L1i size: 1kB L1d size: 1kB
L2 Cache	ways: {2,4,8} tag latency: 20 data latency: 20 response latency: 20 MSHRs: 20 targets per MSHR: 12 write buffers: 8 type: {fully associative, set associative, skewed associative, ZCache} size: {4kB,8kB,16kB,64kB}
SPEC Benchmarks	bwaves_s,cactuBSSN_s,lbm_s,wrf_s,cam4_s, pop2_s,imagick_s,nab_s,fotonik3d_s,specrand_fs, perlbench_s,gcc_s,mcf_s,omnetpp_s,xalancbmk_s, x264_s,deepsjeng_s,exchange2_s,xz_s

V. Results and Discussion

Table III summarizes the average miss rate and percentage miss rate reduction for ZCache, set associative, and skewed associative caches across all benchmarks. These numbers are averaged across all cache sizes and associativity. Just by glancing at these numbers, one can make a case for the benefits offered by ZCache. The following subsections will look at these numbers in greater detail.

TABLE III: Miss rate impact across individual benchmarks

Benchmark_name	Avg_MissRate_ZCache	Avg_MissRate_Skewed	Avg_MissRate_Set	MissRate Reduction % ZCache/Set	MissRate Reduction % ZCache/Skewed
bwaves_s	0.640	0.645	0.816	-21.606	-0.884
cactuBSSN_s	0.256	0.257	0.288	-11.106	-0.343
cam4_s	0.413	0.409	0.441	-6.307	0.962
deepsjeng_s	0.931	0.931	0.932	-0.027	0.014
exchange2_s	0.220	0.217	0.252	-12.828	1.646
fotonik3d_s	0.432	0.419	0.436	-0.965	2.977
gcc_s	0.380	0.377	0.407	-6.668	0.648
imagick_s	0.018	0.021	0.052	-64.581	-11.122
lbm_s	0.996	0.996	0.996	-0.005	0.000
mcf_s	0.295	0.286	0.328	-10.327	3.065
nab_s	0.376	0.365	0.400	-5.913	3.064
omnetpp_s	0.294	0.294	0.314	-6.401	-0.093
perlbench_s	0.462	0.462	0.488	-5.245	-0.017
pop2_s	0.345	0.344	0.370	-6.709	0.345
specrand_fs	0.420	0.414	0.468	-10.293	1.337
wrf_s	0.376	0.376	0.392	-4.168	-0.123
x264_s	0.364	0.364	0.374	-2.667	-0.006
xalancbmk_s	0.368	0.369	0.404	-8.776	-0.378
xz_s	0.842	0.842	0.846	-0.458	0.070

A. Impact of Cache Parameters on Miss Rate

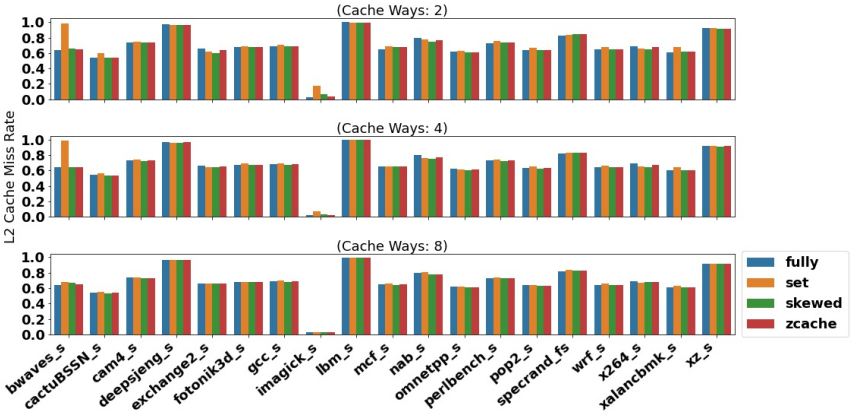
We vary L2 cache parameters for several SPEC 2017 benchmarks as described in Table II, and measure the L2 cache miss rate for 20 million instructions. The miss rates for the various cache configurations and benchmarks are shown in Fig. 3. Please note that the associativity of the fully associative caches does not change within a cache size and is presented in the figure as a benchmark for other caches.

The miss rate significantly decreases across all cache configurations for most benchmarks as the cache size increases. Increasing cache size allows the cache to hold a larger portion of the working set, resulting in reduced capacity misses and overall miss rate. 'deepsjeng_s','lbm_s' and 'xz_s' have very large working sets [5] and can not be accommodated by our largest caches. Consequently, they have a miss rate of nearly one across all configurations.

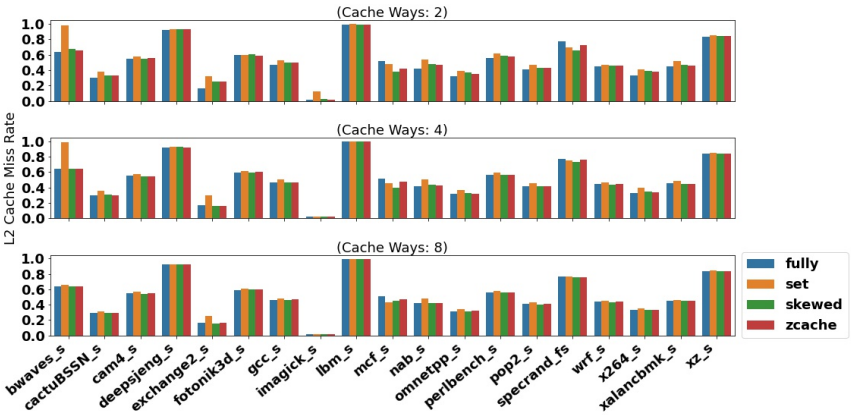
Compared to varying the cache size, increasing the number of cache ways and varying the cache type have minor impacts on the miss rate. Fully associative caches have very similar miss rates compared to set associative caches, skewed associative caches, and ZCaches for the majority of cache configurations. Since fully associative caches do not have conflict misses, the similarity in miss rates between fully associative and other caches indicates that capacity and compulsory misses constitute the bulk of the misses. Therefore varying the number of cache ways and cache type has little effect on the miss rate since both impact conflict misses. We perform further analysis to measure the impact of cache type on a smaller set of cache configurations and benchmarks where capacity misses make up a significant portion of the total number of misses.

B. Impact of Cache Parameters on IPC

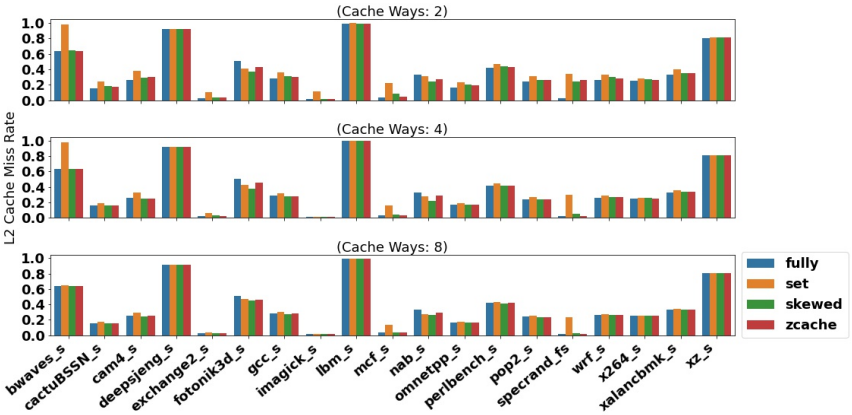
IPC for the various cache configurations and benchmarks from Table II are shown in Fig. 4. The IPC across all configurations is smaller than 0.2. We use small L1 instruction and data caches to amplify L2 cache accesses and the impact of L2 cache parameters on its miss rate. Consequently, the high number of L1 cache misses bottleneck the IPC for all of our results. The L2 cache configuration has a similar relationship with IPC compared to L2 cache misses, and we perform further analysis to measure the impact of cache type.



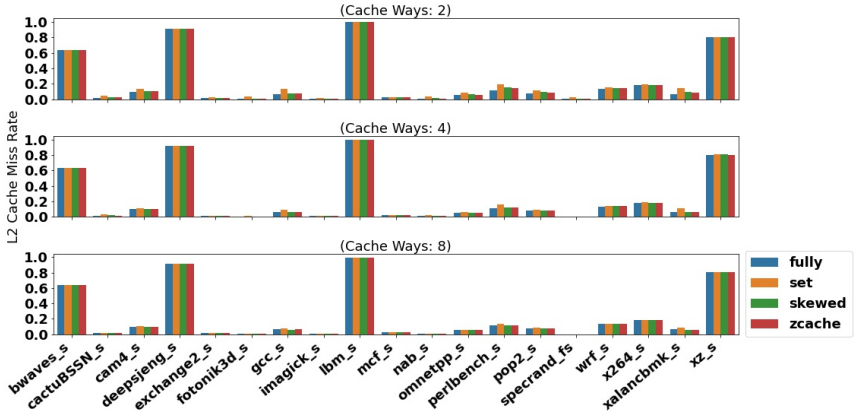
(a) Miss rates for various 4Kb L2 Caches



(b) Miss rates for various 8Kb L2 Caches

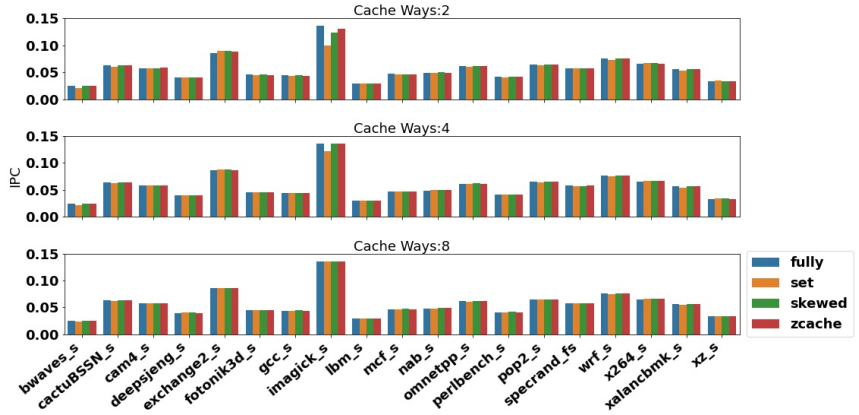


(c) Miss rates for various 16Kb L2 Caches

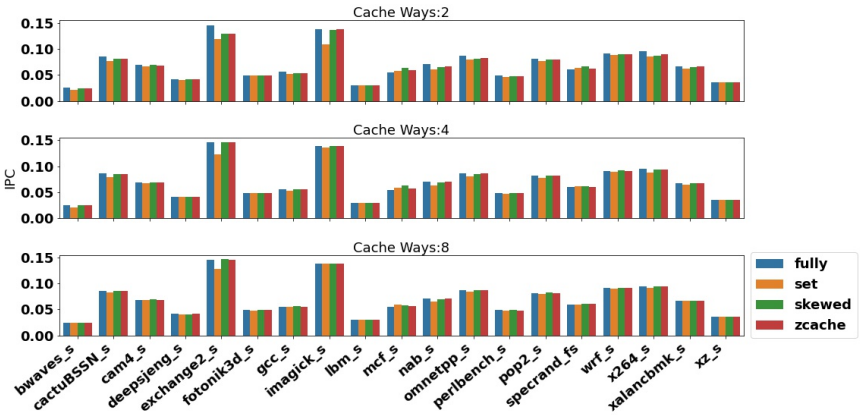


(d) Miss rates for various 64Kb L2 Caches

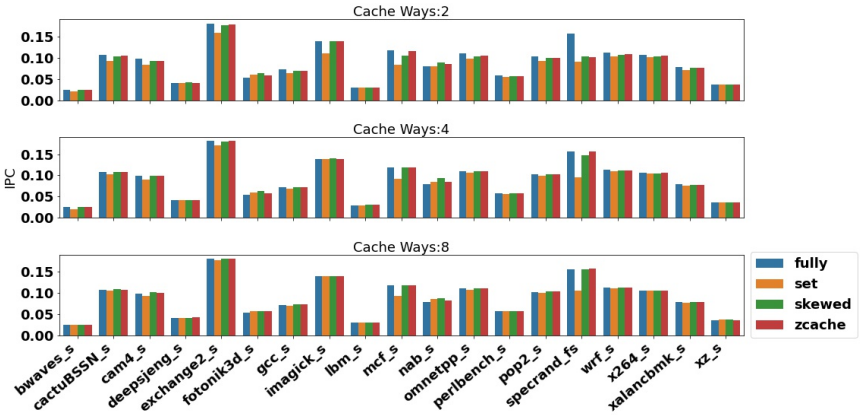
Fig. 3: Miss Rates for Various L2 Caches while Running SPEC 2017 Benchmarks



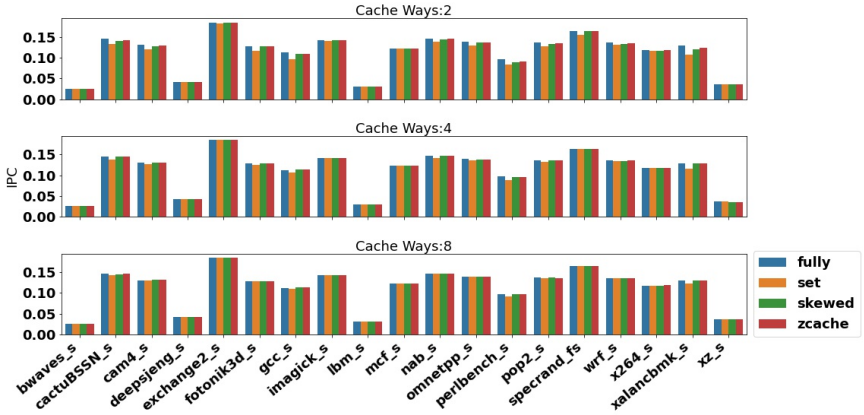
(a) IPC for various 4Kb L2 Caches



(b) IPC for various 8Kb L2 Caches



(c) IPC for various 16Kb L2 Caches



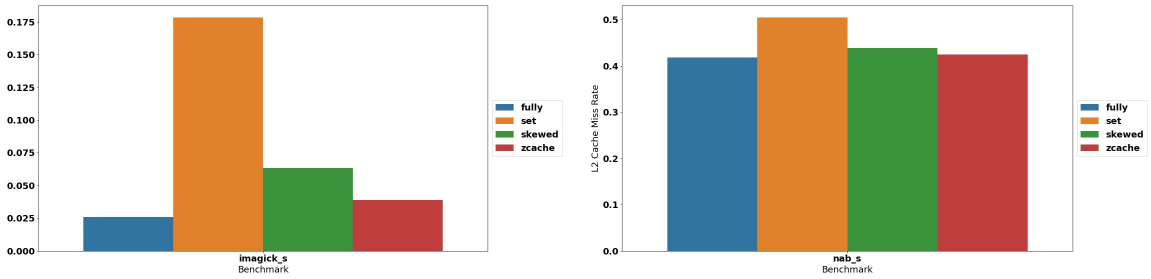
(d) IPC for various 64Kb L2 Caches

Fig. 4: IPC for Various L2 Caches while Running SPEC 2017 Benchmarks

C. *Impact of ZCache on Conflict Misses*

In the above sub-sections, we looked at the impact of ZCache across all benchmarks with varying cache sizes and ways. However, to really understand the difference we need to looked more closely at the results.

The miss rates for fully associative and other types of caches are very similar for a large number of our results, indicating that conflict misses have little to no contribution to the overall miss rate. We extract a subset from our results where fully associative cache performs at least 5% better than set associative cache to observe cases where conflict misses have a significant contribution to the miss rate. Table VII in Appendix contains all cases that satisfy this condition with miss rate and IPC numbers for all four cache types. Figure 5 shows L2 cache miss rate of two such cases.



(a) L2 Miss Rate vs Cache type: Imagick
CacheSize: 4kB Associativity 2

(b) L2 Miss Rate vs Cache type: Nab
CacheSize: 8kB Associativity 4

Fig. 5: Impact of Conflict Misses

The tables below summarize cache performance in the selected subset. For these cases, we can see a clear difference in the performance benefits Z-Cache offers over set associative caches. ZCache reduces miss rate by providing more replacement candidates. However, skewed associative caches perform almost equally better. Skewed associative caches significantly reduce the number of conflict misses such that they constitute a small minority of the overall miss rate, as exhibited by the similarity in miss rates between fully and skewed associative caches. Per Amdahl's Law, reducing the number of conflict misses further with ZCache only has a minor impact on the overall miss rate.

TABLE IV: Cache impact on the selected cases

Cache Type	Avg. IPC	Avg. MissRate
Set Associative	0.079	0.421
Skewed Associative	0.090	0.313
ZCache	0.091	0.307
Fully Associative	0.094	0.288

TABLE V: Z-Cache comparison with Skewed and Set Associative caches

Cache Type	IPC increase	Miss Rate Reduction
ZCache v/s Set Associative	15.58%	27.06%
ZCache v/s Skewed Associative	1.17%	1.98%

TABLE VI: Miss rate impact across individual benchmarks (Conflict Misses)

Benchmark_name	Avg_MissRate_ZCache	Avg_MissRate_Skewed	Avg_MissRate_Set	MissRate Reduction % ZCache/Set	MissRate Reduction % ZCache/Skewed
bwaves_s	0.642	0.650	0.985	-34.875	-1.296
cactuBSSN_s	0.334	0.338	0.392	-14.733	-1.195
cam4_s	0.276	0.272	0.353	-21.867	1.435
exchange2_s	0.151	0.151	0.243	-37.778	-0.016
gcc_s	0.294	0.296	0.339	-13.466	-0.651
imagick_s	0.025	0.034	0.140	-82.056	-26.110
mcf_s	0.038	0.052	0.171	-77.990	-27.443
nab_s	0.436	0.447	0.508	-14.117	-2.383
omnetpp_s	0.289	0.297	0.331	-12.590	-2.705
perlbench_s	0.362	0.367	0.404	-10.441	-1.270
pop2_s	0.344	0.345	0.388	-11.428	-0.409
specrand_fs	0.101	0.107	0.291	-65.398	-5.551
wrf_s	0.286	0.300	0.329	-12.976	-4.384
x264_s	0.356	0.369	0.399	-10.818	-3.586
xalancbmk_s	0.378	0.383	0.433	-12.603	-1.273

VI. Conclusion

In this work, we implement ZCache and analyze its impact on cache and system performance while running SPEC2017 benchmarks. ZCache seeks to improve performance by providing more replacement candidates than set associative caches. We implement ZCache in GEM5 for TimingSimpleCPU and verify its correctness for ARM and X86. We collect results for several benchmarks with varying cache sizes and number of ways to study the impact of ZCache. Firstly, we notice that specific benchmarks 'deepsjeng_s','lbm_s' and 'xz_s' show high miss rate across all cache types and sizes (including fully associative caches) indicating that these benchmarks have bigger working set and memory footprint which is confirmed by [5]. At the surface level, it seems that ZCache has a similar performance to set associative caches. However, when we look closely at cases with more conflict misses, we notice the true impact of ZCache, which reduces the miss rate by $\sim$27% compared to set associative caches with the same number of ways and cache size. Although ZCache significantly improves miss rate compared to set associative caches, it only offers a 1.98% improvement compared to skewed associative caches. We attribute this to the diminishing impact of reducing conflict misses on the overall miss rate per Amdahl's law.

References

[1] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 187–198.

[2] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 169–178. [Online]. Available: https://doi.org/10.1145/165123.165152

[3] F. Bodin and A. Seznec, "Skewed associativity enhances performance predictability," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 265–274. [Online]. Available: https://doi.org/10.1145/223982.224437

[4] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 364–373. [Online]. Available: https://doi.org/10.1145/325164.325162

[5] S. Singh and M. Awasthi, "Memory centric characterization and analysis of spec cpu2017 suite," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 285–292. [Online]. Available: https://doi.org/10.1145/3297663.3310311

VII. Appendix

A. *Conflict Misses*

TABLE VII: All cases with 5% higher conflict miss in set-associative caches as compared to fully-associative caches

Benchmark name	CacheSize (kB)	Cache Associativity	MissRate _Fully	MissRate _ZCache	MissRate _Skewed	MissRate _Set	IPC _Fully	IPC _ZCache	IPC _Skewed	IPC _Set
bwaves_s	4	2	0.641	0.647	0.662	0.988	0.025	0.025	0.025	0.021
bwaves_s	4	4	0.641	0.641	0.644	0.991	0.025	0.025	0.025	0.021
bwaves_s	8	2	0.638	0.654	0.676	0.984	0.025	0.025	0.024	0.021
bwaves_s	8	4	0.638	0.638	0.638	0.985	0.025	0.025	0.025	0.021
bwaves_s	16	2	0.635	0.636	0.646	0.982	0.025	0.025	0.025	0.021
bwaves_s	16	4	0.635	0.635	0.635	0.982	0.025	0.025	0.025	0.021
cactuBSSN_s	4	2	0.541	0.537	0.535	0.596	0.063	0.063	0.064	0.060
cactuBSSN_s	8	2	0.297	0.329	0.336	0.378	0.086	0.081	0.081	0.076
cactuBSSN_s	8	4	0.297	0.298	0.302	0.353	0.086	0.085	0.085	0.079
cactuBSSN_s	16	2	0.157	0.172	0.180	0.241	0.107	0.104	0.103	0.093
cam4_s	16	2	0.258	0.301	0.294	0.384	0.099	0.092	0.093	0.083
cam4_s	16	4	0.258	0.251	0.250	0.323	0.099	0.100	0.100	0.089
exchange2_s	8	2	0.165	0.251	0.254	0.319	0.145	0.128	0.129	0.119
exchange2_s	8	4	0.165	0.159	0.160	0.294	0.145	0.146	0.146	0.122
exchange2_s	8	8	0.165	0.162	0.153	0.254	0.145	0.146	0.147	0.128
exchange2_s	16	2	0.025	0.032	0.037	0.103	0.181	0.179	0.178	0.159
gcc_s	8	2	0.466	0.500	0.500	0.530	0.056	0.053	0.053	0.052
gcc_s	16	2	0.283	0.305	0.309	0.358	0.072	0.069	0.069	0.064
gcc_s	64	2	0.065	0.076	0.078	0.130	0.112	0.109	0.108	0.096
imagick_s	4	2	0.026	0.039	0.063	0.178	0.136	0.131	0.124	0.099
imagick_s	8	2	0.019	0.020	0.023	0.126	0.138	0.138	0.137	0.109
imagick_s	16	2	0.016	0.016	0.016	0.115	0.139	0.139	0.139	0.111
mcf_s	16	2	0.032	0.048	0.082	0.219	0.118	0.115	0.105	0.083
mcf_s	16	4	0.032	0.033	0.040	0.155	0.118	0.118	0.118	0.091
mcf_s	16	8	0.032	0.032	0.033	0.139	0.118	0.118	0.118	0.093
nab_s	8	2	0.418	0.465	0.480	0.542	0.071	0.067	0.065	0.061
nab_s	8	4	0.418	0.424	0.439	0.505	0.071	0.070	0.069	0.064
nab_s	8	8	0.418	0.420	0.421	0.477	0.071	0.071	0.070	0.065
omnetpp_s	8	2	0.317	0.355	0.366	0.388	0.087	0.082	0.081	0.079
omnetpp_s	8	4	0.317	0.321	0.328	0.371	0.087	0.086	0.085	0.081
omnetpp_s	16	2	0.167	0.192	0.198	0.234	0.110	0.105	0.104	0.098
perlbench_s	8	2	0.562	0.581	0.584	0.620	0.049	0.048	0.047	0.046
perlbench_s	64	2	0.115	0.143	0.149	0.189	0.097	0.091	0.090	0.083
pop2_s	8	2	0.414	0.430	0.429	0.470	0.082	0.080	0.080	0.076
pop2_s	16	2	0.240	0.257	0.261	0.306	0.103	0.100	0.099	0.093
specrand_fs	16	2	0.022	0.258	0.247	0.341	0.157	0.101	0.103	0.091
specrand_fs	16	4	0.022	0.023	0.047	0.298	0.157	0.156	0.147	0.096
specrand_fs	16	8	0.022	0.021	0.025	0.233	0.157	0.157	0.155	0.105
wrf_s	16	2	0.262	0.286	0.300	0.329	0.113	0.109	0.107	0.103
x264_s	8	2	0.330	0.378	0.393	0.406	0.095	0.089	0.088	0.086
x264_s	8	4	0.330	0.334	0.346	0.393	0.095	0.094	0.093	0.088
xalancbmk_s	4	2	0.608	0.618	0.621	0.676	0.056	0.056	0.056	0.053
xalancbmk_s	8	2	0.450	0.464	0.468	0.519	0.067	0.066	0.065	0.062
xalancbmk_s	16	2	0.330	0.349	0.352	0.396	0.078	0.076	0.076	0.072
xalancbmk_s	64	2	0.063	0.082	0.091	0.141	0.129	0.123	0.121	0.108