Decoupling Ways and Associativity in gem5

Jason Panelli
panelli@ucla.edu
University of California - Los Angeles

Samuel Lee
samuellee@cs.ucla.edu
University of California - Los Angeles

Emily Kao
emilykao@ucla.edu
University of California - Los Angeles

Abstract

Caches are now more than ever being utilized for high-capacity applications that require high associativity. In such applications, we commonly see the number of ways and associativity increased to accommodate for large data movement. gem5 in particular is a simulator system growing in popularity in projects related to computer-system architecture. In fact, it is used in a variety of applications such as architecture design, industry research, and teaching. gem5 is widely used, and there are many projects that would require a high associativity and thus a higher number of ways. We present zcache, a cache designed by Stanford graduate students that enables caches to have high associativity with a low number of ways. zcache utilizes an algorithm that relies on the number of replacement candidates for more associativity. This decouples ways from associativity, ultimately leading to higher performance and better energy efficiency. In this paper, we propose an extension of gem5 to support the zcache design, which can then be used in relevant applications.

In particular, we hypothesize several benefits from the zcache design. We expect to see a performance increase with zcache over most workloads, and we also show that associativity is better than the default associative-set cache in gem5. We run several simulations in order to show the significance of the decoupling of ways and associativity and how it is a necessity in gem5 cache configurations.

1 Introduction

In traditional caching, associativity and the number of ways are equivalent. Cache associativity is managed simply through the number of ways in the cache. This comes at the cost of energy trade-offs with respect to supporting more ways and can additionally increase hit latency and lead to more expensive hardware. zcache is a solution that decouples associativity from the number of ways as a means to improve the negative side effects of these trade-offs. If higher associativity can be achieved with the same number of ways, then the cache can gain the advantages of higher associativity (namely lower miss rate) without the complex hardware needs of a high number of ways. zcache cleverly implements this by shuffling data around in the cache when necessary in order to provide more replacement candidates.

The gem5 simulator is a platform for computer architecture research that is used to simulate and test new microarchitecture ideas. In its current form, it has a number of cache replacement policies and a skew cache implemented but does not implement any form of zcache. The aim of this project is to add the core ideas of zcache to gem5. In order to so, we extend the skew cache and require support for gathering eviction candidates, selecting a candidate to evict, and moving data in the cache to accommodate new data. Doing so adds the main ideas from zcache.

2 Related Work

Our project aims to modify the gem5 simulator, a platform for computer architecture research, to support the zcache, a novel cache design developed by Daniel Sanchez and Christos Kozyrakis of the Electrical Engineering Department at Stanford University [7]. In general cache design, it is accepted that the associativity of a cache is directly related to the number of locations per way. However, in the development of zcache, the authors observe that the associativity is dependent on the number of replacement candidates during an eviction rather than the number of locations per way [7]. The design of zcache has two main inspirations. First, zcache is similar to skew-associative caches in that it utilizes different hash functions to access each way [1]. According to the research of Francois Bodin and Andre Seznec in their paper "Skewed associativity enhances performance predictability", skewed caches are much less sensitive to parameters such as array size compared to direct-mapped caches and set-associative caches, which results in more consistent and stable performance [1]. Second, zcache implements a similar technique as cuckoo hashing during replacements by exploiting multiple hash functions to find a non-conflicting location in another way when there is a conflict [6]. Cuckoo hashing is a technique introduced by Rasmus Pagh and Flemming Friche Rodler in the creation of their own dictionary data structure that has a worst case constant lookup time while using a much simpler design [6]. A combination of these two designs in zcache yields increased performance and energy efficiency, which will be investigated in our own implementation of zcache in gem5.

zcache's novel design is able to increase the associativity of the cache, however a good replacement policy is also necessary to increase cache performance. The following is some research into other replacement policies that have better performance than LRU: pseudo-LIFO that is a series of replacement policies called dead block prediction LIFO, probabilistic escape LIFO, and probabilistic counter LIFO [2], Thread-Aware Dynamic Insertion Policy [3], Re-reference Interval Prediction [4], and Promotion/Insertion Pseudo-Partitioning (PIPP) [9].

3 Methods

In order to implement zcache in gem5, we had to inspect several different modules of gem5. Specifically, gem5 memory is organized into a cache directory that is organized into modules such as indexing policies, prefetchers, replacement policies, and more. After three different implementations of zcache involving the base cache, existing indexing policies, and existing prefetching methods, we finally wrote a successful copy of the zcache design and algorithm in our own submodule with a different indexing policy and a different prefetching method. This involved the addition of fields to replacement objects, new functions to override the base cache, and a swap structure to handle our relocations, which will be detailed in the subsections below. In order to analyze how to extend gem5, we developed a diagram of gem5's cache structure. As shown in Figure 1, we can see that caches in gem5 are created into entries. For insertions of data, each entry can be mapped to the address of the data, tag, set, and way. The set of an entry is calculated by the quotient of the index divided by the associativity, and the way of an entry is calculated by the modulus of the index and the associativity. With this insight, gem5 cache entries can be utilized to implement the replacement candidates algorithm as well as the relocation algorithm.

3.1 Skew Cache Indexing Policy

As mentioned by the zcache paper, the zcache design shares many similar functions as the skew-associative cache [7]. In particular, zcache takes advantage of hashing each way by different hash functions. Fortunately, gem5's prefetcher module already has a skew cache that implements up to 8 hash functions. Our zcache implementation utilizes the skew cache hashing functions as well as the algorithm to retrieve potential candidates. One additional functionality in our version of the skew cache is the addition of a parent index in order to track the tree nodes for our BFS relocation algorithm. The class ReplaceableEntry is an entry in a 2d table-like structure with replacement functionality for replacement policy use [5]. As shown below in the code snippet, we added the parent index to this entry.

```
class ReplaceableEntry
{
  protected:
    uint32_t _set;
    uint32_t _way;
    uint32_t _parent;

    /**
     * Set the set, way, and parent
     *
     * @param set The set of this entry.
     * @param way The way of this entry.
     * @param parent The parent of this entry.
     */
    virtual void
    setPosition(const uint32_t set,
                const uint32_t way,
                const uint32_t parent)
    {
      _set = set;
      _way = way;
      _parent = parent;
    }
    ........
};
```

3.2 Replacement Candidates Algorithm

The replacement candidates algorithm is one of the two core algorithms of zcache. Getting this algorithm to work was essential for a gem5 addition. Current algorithms to find replacement candidates are typically limited to one level of replacement candidates. The original zcache design proposes a breadth-first graph walk to get additional sets of candidates [7]. The current implementation in gem5 is a breadth-first graph walk limited to two levels of candidates. This walk is done in three major steps - calculating the entry index, generating new possible entries, and adding new candidates. As seen below in the code snippet, the first set of selected entries is expanded to a second level of candidates.

```
for (const auto &location : selected_entries)
{
    ...
    // 1) Calculate entry index
    unsigned int entryId = (location->getSet() *
              ssociativity) + location->getWay();
    ...
    // 2) Generate new possible entries
    const std::vector<ReplaceableEntry *>
               extra_entries =
               indexingPolicy->
               getPossibleEntries(entryAddr);
    ...
    // 3) Add new candidates
    for (uint32_t i = 0; i < associativity; i++)
    {
        ...
        // Avoid Repeats
        if (entryId != entryId2)
        {
            numCandidates++;
        }
    }
}
```

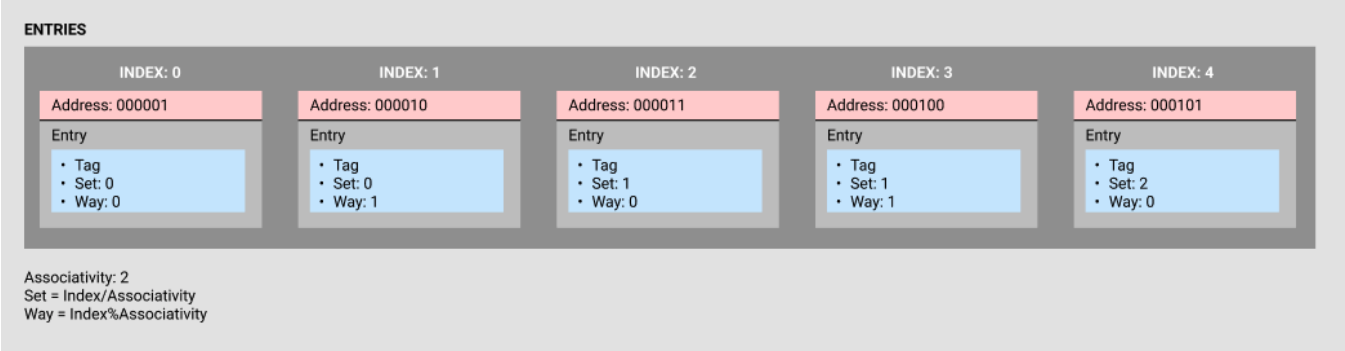

Figure 1. Representation of entries and addressing in gem5.

3.3 Relocation Algorithm

After the accumulation of candidates, the next step of zcache's design takes place in relocations. The walk finishes, and the replacement candidate is now evicted. In the same cache prefetch module, we implemented an algorithm that relocates the ancestors in the tree in order to accept the incoming block as described in the zcache paper [7]. This replacement process takes place with 2 important parts. The first portion is presented in the code below.

```
unsigned int victimIndex =
    (victim->getSet() * associativity)
    + victim->getWay();
int index = victimIndex;
int swapIndex = 0;
while (index >= 0)
{
    Entry *currEntry = &entries[index];
    swapArrSet[swapIndex] = currEntry->getSet();
    swapArrWay[swapIndex] = currEntry->getWay();
    index = currEntry->getParent();
    swapIndex++;
}
```

The first part involves filling a swap array with the set and way according to the new index that it would be assigned to. The next index to swap comes from the parent of the current Entry. This is performed until all the necessary relocations are accounted for.

```
for (uint32_t i = 0; i < swapLen - 1; i++)
{
    int index1 = (swapArrSet[i] * associativity)
                            + swapArrWay[i];
    int index2 = (swapArrSet[i + 1] * associativity)
                            + swapArrWay[i + 1];
    entries[index1] = entries[index2];
}
```

The second portion of the algorithm performs all the swaps in the entries array, and it calculates all the new indices as well. This is then set to the new positions, and the final incoming block is set to the last remaining entry.

3.4 Cache Configuration in gem5

While we were able to get a working zcache design algorithm to find replacement candidates as well as to perform relocations, we found difficulty in figuring out the cache configuration options in gem5. In particular, we hoped to modify the L1 and L2 cache options to provide zcache as an option in gem5. However, we found little to no success in finishing this portion of the project given our time constraints. Instead, we hardcoded zcache arrays into associative sets that were being used throughout gem5. Ideally, this project would be able to toggle the use of zcache for the L1 and L2 caches, and then we could use spec2017 [8] to evaluate the performance of zcache. This will be saved for future work on the gem5 extension.

4 Methodology

4.1 Testing Configuration and Overview

To evaluate the performance of our implementation of zcache in gem5, we use the default configuration of gem5 in homework 3 and compare the performances of the default cache and the zcache. The default configuration has the following parameters: 8GB of physical memory size, the DerivO3CPU, 64kB L1D, 64kB L1I, 2MB L2, 2GHz frequency, and LRU replacement policy. We run these two configurations with the following three spec2017 benchmarks: lbm_s, imagick_s, and nab_s. While we would like to compare the default cache and zcache across all provided benchmarks, due to a lack of time, we chose these three benchmarks. After running the benchmarks, it generates a stats.txt file. We parse through the file and extract the following statistics:

simSeconds, system.cpu.dcache.overallMissRate::total, system.cpu.dcache.overallAvgMissLatency::total, system.l2.overallMissRate::total, and system.l2.overallAvgMissLatency::total. simSeconds is the number of seconds simulated, system.cpu.dcache.overallMissRate::total is the miss rate for overall accesses in the L1D cache, system.cpu.dcache.overallAvgMissLatency::total is the average overall miss latency in Cycles/Count in the L1D cache, system.l2.overallMissRate::total is the miss rate for overall accesses in the L2 cache, and system.l2.overallAvgMissLatency::total is the average overall miss latency in Cycles/Count in the L2 cache.

lbm_s, imagick_s, and nab_s are benchmarks in SPEC CPU 2017 Floating Point Testing Suites. Spec CPU 2017 focuses on compute intensive performance, specifically evaluating the performance of the Processors, Memory, and Compilers [8]. The benchmarks we've chosen are specifically floating point compute intensive programs. lbm_s implements the "Lattice Boltzmann Method" to simulate the behavior of fluids. imagick_s is a program that takes in a series of operations to perform on an image. nab_s is based on a molecular modeling application called Nucleic Acid Builder (NAB) that performs a series of floating point operations [8].

5 Evaluation

As mentioned before, the results were taken in an environment where zcache was simply hardcoded to substitute for associative sets in gem5. These results are not representative of a zcache change in L1 and L2 caches. In Table ??, we can see that all of the results are identical. While we were able to successfully implement zcache, we would like to be able to configure L1 and L2 caches to use the design in the future. Once this is implemented, we expect to see a significant gain in performance and the ability to decrease miss rate by increasing associativity without incurring the overhead cost of increasing ways. While these are not the results that were hoped for, we were able to conclude that a lot of the current gem5 implementation that uses associative sets are not critical to the performance, which explains why we are unable to see any performance gains from the results.

6 Conclusion

6.1 Overview

We have implemented the zcache core algorithms and structures so that gem5 can be extended to use caches where ways and associativity are decoupled. We found the gem5 structure and modules to be non-trivial, and researching the use of cache entries was crucial to our understanding of the addressing and structures required to implement the zcache design. We achieved our goals of decoupling ways and associativity in gem5, and we look forward to what work is to come in this extension.

zcache Configuration	Default configuration
(nab-s) simSeconds: 0.040037	(nab-s) simSeconds: 0.040037
(nab-s) system.cpu.dcache overallMissRate::total: 0.002515	(nab-s) system.cpu.dcache overallMissRate::total: 0.002515
(nab-s) system.l2 overallMissRate::total: 0.046818	(nab-s) system.l2 overallMissRate::total: 0.046818
(lbm-s) simSeconds: 0.135084	(lbm-s) simSeconds: 0.135084
(lbm-s) system.cpu.dcache overallMissRate::total: 00.040215	(lbm-s) system.cpu.dcache overallMissRate::total: 00.040215
(lbm-s) system.l2 overallMissRate::total: 1	(lbm-s) system.l2 overallMissRate::total: 1
(imagick-s) simSeconds: 0.026700	(imagick-s) simSeconds: 0.026700
(imagick-s) system.cpu.dcache overallMissRate::total: 0.001163	(imagick-s) system.cpu.dcache overallMissRate::total: 0.001163
(imagick-s) system.l2 overallMissRate::total: 0.964345	(imagick-s) system.l2 overallMissRate::total: 0.964345

Table 1. Results for nab-s, lbm-s, and imagick-s on zcache and default configs

6.2 Future Work

While our implementation achieves the core functionality of zcache, a fully fledged zcache and extensions of zcache require more support. Namely, our implementation ignores replacement policy and just focuses on the aspect of zcache involved with increasing replacement candidates and block relocation. The original paper relays that replacement policy can become more difficult in the zcache skewed cache setting because it is impossible to maintain ordering of a set (and a set itself simply is not a useful zcache/skew cache concept) [7]. This is problematic for policies like LRU that can be efficiently implementing by exploiting set ordering. The paper describes two options - full LRU and bucketed LRU - where full LRU stores a timestamp for each block and uses the timestamp in order to select a victim and bucketed LRU stores a condensed time counter that is periodically incremented to save space. Neither of these replacement policies were considered in our implementation, but could greatly improve our zcache implementations performance.

Another improvement to our zcache would be to simply search deeper in the cache. In particular, we provide strictly two levels of search into the zcache replacement candidates

and do not proceed deeper than this. In practice, zcache may search arbitrarily deep. A decent extension to our implementation would be to add an additional parameter for this practice as an associativity flag to give more levels of control to the program execution and simulation. If this is achieved, a user can investigate the trade-offs of including more and less levels of zcache in the replacement candidates.

One glaring issue with zcache as a whole is that the walking process and shifting in the cache may introduce additional latency in a cache miss. The authors of the zcache paper account for this by saying that the walk should run while the data that was missed is searched for in the rest of the memory hierarchy [7]. This, however, introduces some additional complexities. Namely, while the walk should complete before the miss is recovered, there is still a race condition that must be accounted for here. Another concurrency issues is also exposed here where the data that is "walking" must also be locked throughout the walk and the walk must complete as an atomic operation. None of these concurrency issues are addressed in our gem5 simulator implementation and thus we may be carrying the additional latency associated with performing the walk on a miss.

The current skewed cache implementation in gem5 also only supports hash functions for 8 ways. Beyond this, the authors of this skew cache implementation provide no guarantees about the efficiency of additional ways [5]. Since our solution is built on top of this skewed cache solution, our zcache can only support 8 ways. We did not see this as worth extending in the scope of our project, but did hypothesize that it could be improved with smarter hash functions that can be generated for each way. For a simulator implementation, the use of pseudo-random functions that take a seed as input and output a pseudo-random generator are used in cryptography to generate hash functions. We hypothesize that this technique can be used as a means to generate hash functions for an unlimited number of ways. This alone is an extension worthy of its own project with its own scope, but the idea here is that smarter hashing can improve gem5's skew cache and hash function implementation. A simpler extension in this regard would simply be to provide more hash functions than are currently available.

Lastly, per the original zcache paper itself, two additional optimizations to zcache are to avoid repeats in the walk and different walking strategies. Avoiding repeats will lead to improvements after the replacement candidate tree is searched deeper than we currently search (two levels) but can avoid redundant cycles and can be implemented in a simulator most easily with a hash table for efficient look ups. This is likely more difficult in hardware. Different walking strategies can also be an additional parameter to zcache and trade-offs can be investigated similar to replacement policy. Namely, instead of a breadth-first-search a zcache user may wish to use depth-first-search or a more optimized tree search to more efficiently find replacement candidates.

6.2.1 Future Work with Respect to Trade-offs. It is worth discussing these extensions with regard to their trade-offs in performance and energy efficiency. Firstly, the extension with regards to replacement policy can improve performance at the cost of consuming more energy and space by requiring support for these extra operations to store and track timestamps. The bucketed LRU reduces the costs of these trade-offs but still requires additional space and operations.

The hash table extensions are not particularly relevant beyond the simulator abstraction, but, typical for cache, efficient hash functions will lead to the most efficient caches. It is best to select the best hash function for a particular use case and doing so can reduce collisions without considering zcache at all and make for better utilization of the cache. Additionally, if a cache function is overly complex, its operations may expend more energy and consume more time.

Next, we observe that searching in the cache is the major feature and optimization of zcache and comes with a number of trade-offs between performance and energy. We can combine our "tree search" optimizations to gain a full understanding of this aspect. We see the depth of the tree, the selection of search algorithms, and the decision of whether we want to eliminate re-visiting the same nodes all have performance and energy trade-offs.

First, we note that searching deeper in the cache "tree" will cost more energy but produce more replacement candidates, so a deeper tree will cost more energy but also improve associativity. We might want to consider how this stacks against the additional ways costing more energy to improve associativity.

The zcache authors also note that using a DFS algorithm reduces the storage required in searching through the tree as it does not require state to revisit previous levels of the tree like BFS [7]. DFS only needs to track the nodes in its current path. This comes at the cost of eliminating the opportunity to avoid repeats in the walk as now only the nodes on the DFS path will be available. Maybe a carefully selected set of hash functions can limit this as well. While this DFS algorithm may increase time complexity because it is likely to repeat many node visits, it can greatly reduce space complexity and as such might consume less energy because it requires less structural support. Other search algorithms will also have their own trade-offs when applied to the zcache walk.

Finally, in general, avoiding re-visiting nodes will require additional structures to track the already-visited nodes and thus this has a built-in performance and energy trade-off as well. In short, it is likely necessary to control each of these features with flags and find the right trade-off through investigation. Looking at different flags for tree depth, tree search algorithms, and revisiting nodes can allow for analysis on performance/energy-efficiency trade-offs in these optimizations.

7 Statement of Work

Samuel Lee: Implemented and researched zcache replacement algorithm, zcache relocation algorithm, and gem5 support for zcache. Wrote Abstract, Methods, Methodology, Evaluation, and part of Conclusion. Ran tests and evaluated results.

Emily Kao: Researched and studied inspirations of zcache as well as potential extensions for zcache. Wrote the Related Works and Methodology section. Wrote a running and parsing script for simulation runs.

Jason Panelli: Created initial project plan, researched zcache algorithm and worked out a more feasible zcache subset, worked on zcache's movement of data through the cache. Wrote Introduction and Future Work, including plenty of studying to find appropriate future work.

References

[1] F. Bodin and A. Seznec. 1995. Skewed associativity enhances performance predictability. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 265–274.

[2] Mainak Chaudhuri. 2009. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 401–412. https://doi.org/10.1145/1669112.1669164

[3] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 208–219.

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (jun 2010), 60–71. https://doi.org/10.1145/1816038.1815971

[5] Jason Lowe-Power. 2022. *gem5 Indexing Policies*. Retrieved March 18, 2022 from https://www.gem5.org/documentation/general_docs/memory_system/indexing_policies/#:~:text=Gem5%20implements%20skewed%20caches%20as,generated%20hash%20function%20is%20used.

[6] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (may 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[7] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 187–198. https://doi.org/10.1109/MICRO.2010.20

[8] SPEC. 2022. *Spec CPU 2017 overview*. Retrieved March 18, 2022 from https://www.spec.org/cpu2017/Docs/overview.html#benchmarks

[9] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 174–183. https://doi.org/10.1145/1555754.1555778