*Appears in the Proceedings of the 43rd Annual IEEE/ACM Symposium on Microarchitecture (MICRO-43), 2010*

# The ZCache: Decoupling Ways and Associativity

Daniel Sanchez and Christos Kozyrakis
Electrical Engineering Department
Stanford University
Email: {sanchezd, kozyraki}@stanford.edu

***Abstract*—The ever-increasing importance of main memory latency and bandwidth is pushing CMPs towards caches with higher capacity and associativity. Associativity is typically improved by increasing the number of ways. This reduces conflict misses, but increases hit latency and energy, placing a stringent trade-off on cache design. We present the *zcache*, a cache design that allows much higher associativity than the number of physical ways (e.g. a 64-associative cache with 4 ways). The zcache draws on previous research on skew-associative caches and cuckoo hashing. Hits, the common case, require a single lookup, incurring the latency and energy costs of a cache with a very low number of ways. On a miss, additional tag lookups happen off the critical path, yielding an arbitrarily large number of replacement candidates for the incoming block.**

**Unlike conventional designs, the zcache provides associativity by increasing the number of replacement candidates, but not the number of cache ways. To understand the implications of this approach, we develop a general analysis framework that allows to compare associativity across different cache designs (e.g. a set-associative cache and a zcache) by representing associativity as a probability distribution. We use this framework to show that for zcaches, associativity depends only on the number of replacement candidates, and is independent of other factors (such as the number of cache ways or the workload). We also show that, for the same number of replacement candidates, the associativity of a zcache is superior than that of a set-associative cache for most workloads. Finally, we perform detailed simulations of multithreaded and multiprogrammed workloads on a large-scale CMP with zcache as the last-level cache. We show that zcaches provide higher performance and better energy efficiency than conventional caches without incurring the overheads of designs with a large number of ways.**

## I. Introduction

As Moore's law enables chip-multiprocessors (CMPs) with tens and hundreds of cores [22, 40], the limited bandwidth, high latency, and high energy of main memory accesses become an important limitation to scalability. To mitigate this bottleneck, CMPs rely on complex memory hierarchies with large and highly associative caches, which commonly take more than 50% of chip area and contribute significantly to static and dynamic power consumption [29, 43].

The goal of this work is to *improve the efficiency of cache associativity*. Higher associativity provides more flexibility in block (re)placement and allows us to utilize the limited cache capacity in the best possible manner. Last-level caches in existing CMPs are already highly associative and the trend is to increase the number of ways with core count. Moreover, several architectural proposals rely on highly associative caches. For example, many designs for transactional memory and thread-level speculation [13, 19], deterministic replay [42], event monitoring and user-level interrupts [8, 34], and even memory consistency implementations [12] use caches to buffer or pin specific blocks. Low associativity makes it difficult to buffer large sets of blocks, limiting the applicability of these schemes or requiring expensive fall-back mechanisms.

Conventional caches improve associativity by increasing the number of physical ways. Unfortunately, this also increases the latency and energy cost of cache hits, placing a stringent trade-off on cache design. For example, this trade-off limits the associativity of first-level caches in most chips to two or four ways. For last-level caches, a 32-way set-associative cache has up to 3.3× the energy per hit and is 32% slower than a 4-way design. Most alternative approaches to improve associativity rely on *increasing the number of locations* where a block can be placed (with e.g. multiple locations per way [1, 10, 37], victim caches [3, 25] or extra levels of indirection [18, 36]). Increasing the number of possible locations of a block ultimately increases the energy and latency of cache hits, and many of these schemes are more complex than conventional cache arrays (requiring e.g. heaps [3], hash-table-like arrays [18] or predictors [10]). Alternatively, hashing can be used to index the cache, spreading out accesses and avoiding worst-case access patterns [26, 39]. While hashing-based schemes improve performance, they are still limited by the number of locations that a block can be in.

In this paper, we propose a *novel cache design that achieves arbitrarily high associativity with a small number of physical ways*, breaking the trade-off between associativity and access latency or energy. The design is motivated by the observation that *associativity is the ability of a cache to select a good block to evict on a replacement*. For instance, assuming an access pattern with high temporal locality, the best block to evict is the least recently used one in the entire cache. For a transactional memory system, the best block to evict is one that does not store transactional metadata. A cache that provides a higher quality stream of evicted blocks essentially has higher associativity, regardless of the number of ways it uses and the number of locations each block can be placed in.

Our three main contributions are:

1) We propose *zcache*, a cache design that improves associativity while keeping the number of possible locations (i.e. ways) of each block small. The zcache's design is based on the insight that *associativity is not determined by the number of locations that a block can reside in, but by the number of replacement candidates on an eviction*. Like a skew-associative cache [39], a zcache accesses each way

using a different hash function. A block can be in only one location per way, so *hits, the common case, require only a single lookup*. On a replacement, the zcache exploits that with different hash functions, a block that conflicts with the incoming block can be *moved* to a non-conflicting location in another way instead of being evicted to accommodate the new block. This is similar to cuckoo hashing [35], a technique to build space-efficient hash tables. On a miss, the zcache walks the tag array to obtain additional replacement candidates, evicts the best one, and performs a series of relocations to accommodate the incoming block. This happens off the critical path, concurrently with the miss and other lookups, so it has no effect on access latency.

2) We develop *a novel analysis framework* to understand associativity and compare the associativities of different cache designs independently of the replacement policy. We define associativity as a probability distribution and show that, under a set of conditions, which are met by zcaches, associativity depends only on the number of replacement candidates. Therefore, we prove that the zcache *decouples associativity from the number of ways* (or locations that a block can be in).
3) We evaluate a first use of zcaches at the last-level cache of the CMP's memory hierarchy. Using the analytical framework we show that, for the same number of ways, zcaches provide higher associativity than set-associative caches for most workloads. We also simulate a variety of multithreaded and multiprogrammed workloads on a large-scale CMP, and show that zcaches achieve the benefits of highly-associative caches without increasing access latency or energy. For example, over a set of 10 miss-intensive workloads, a 4-way zcache provides 7% higher IPC and 10% better energy efficiency than a 32-way set-associative cache.

The rest of the paper is organized as follows. Section II gives the necessary background on approaches to increase cache associativity. Section III presents the zcache design. Section IV develops the theoretical framework to understand and analyze associativity. Section V discusses our evaluation methodology, and Section VI presents the evaluation of the zcache as a last-level cache. Section VII discusses additional related work, and Section VIII concludes the paper.

## II. Background on Cache Associativity

Apart from simply increasing the number of ways in a cache and checking them in parallel, there is abundant prior work on alternative schemes to improve associativity. They mainly rely on either using hash functions to spread out cache accesses, or increasing the number of locations that a block can be in.

### A. Hashing-based Approaches

*Hash block address:* Instead of using a subset of the block address bits as the cache index, we can use a better hash function on the address to compute the index. Hashing spreads out access patterns that are otherwise pathological, such as strided accesses that always map to the same set. Hashing slightly increases access latency as well as area and power overheads due to this additional circuitry. It also adds tag store overheads, since the full block address needs to be stored in the tag. Simple hash functions have been shown to perform well [26], and some commercial processors implement this technique in their last-level cache [41].

*Skew-associative caches:* Skew-associative caches [39] index each way with a different hash function. A specific block address conflicts with a fixed set of blocks, but those blocks conflict with other addresses on other ways, further spreading out conflicts. Skew-associative caches typically exhibit lower conflict misses and higher utilization than a set-associative cache with the same number of ways [7]. However, they break the concept of a set, so they cannot use replacement policy implementations that rely on set ordering (e.g. using pseudo-LRU to approximate LRU).

### B. Approaches that Increase the Number of Locations

*Allow multiple locations per way:* Column-associative caches [1] extend direct-mapped caches to allow a block to reside in two locations based on two (primary and secondary) hash functions. Lookups check the second location if the first is a miss and a rehash bit indicates that a block in the set is in its secondary location. To improve access latency, a hit in a secondary location causes the primary and secondary locations to be swapped. This scheme has been extended with better ways to predict which location to probe first [10], higher associativities [45], and schemes that explicitly identify the less used sets and use them to store the more used ones [37]. The drawbacks of allowing multiple locations per way are the variable hit latency and reduced cache bandwidth due to multiple lookups, and the additional energy required to do swaps on hits.

*Use a victim cache:* A victim cache is a highly or fully-associative small cache that stores blocks evicted from the main cache until they are either evicted or re-referenced [25]. It avoids conflict misses that are re-referenced after a short period, but works poorly with a sizable amount of conflict misses in several hot ways [9]. Scavenger [3] divides cache space into two equally large parts, a conventional set-associative cache and a fully-associative victim cache organized as a heap. Victim cache designs work well as long as misses in the main cache are rare. On a miss in the main cache, they introduce additional latency and energy consumption to check the victim cache, regardless of whether the victim cache holds the block.

*Use indirection in the tag array:* An alternative strategy is to implement tag and data arrays separately, making the tag array highly associative, and having it contain pointers to a non-associative data array. The Indirect Index Cache (IIC) [18] implements the tag array as a hash table using open-chained hashing for high associativity. The V-Way cache [36] implements a conventional set-associative tag array, but makes it larger than the data array to make conflict misses rare. Tag indirection schemes suffer from extra hit latency, as they are forced to serialize accesses to the tag and data arrays. Both the IIC and the V-Way cache have tag array overheads of around 2×, and the IIC has a variable hit latency.

Several of these designs both increase cache associativity and propose a new replacement policy, sometimes tailored to the proposed design [3, 18, 36, 39]. This makes it difficult to elucidate how much improvement is due to the higher associativity and how much depends on the better replacement policy. In this work we consider that associativity and replacement policy are separate issues, and focus on associativity.

## III. The ZCache Design

Structurally, the zcache shares many common elements with the skew-associative cache. Each way is indexed by a different hash function, and a cache block can only reside in a single position on each way. That position is given by the hash value of the block's address. Hits happen exactly as in the skew-associative cache, requiring a single lookup to a small number of ways. On a miss, however, the zcache exploits the fact that two blocks that conflict on a way often do not conflict on the other ways to increase the number of replacement candidates. The zcache performs a replacement over multiple steps. First, it *walks* the tag array to identify the set of replacement candidates. It then picks the candidate preferred by the replacement policy (e.g. least recently used block for LRU), and evicts it. Finally, it performs a series of *relocations* to be able to accommodate the incoming block at the right location.

The multi-step replacement process happens while fetching the incoming block from the memory hierarchy, and does not affect the time required to serve the miss. In non-blocking caches, simultaneous lookups happen concurrently with this process. The downside is that the replacement process requires extra bandwidth, especially on the tag array, and needs extra energy. However, should bandwidth or energy become an issue, the replacement process can be stopped early, simply resulting in a worse replacement candidate.

### A. Operation

We explain the operation of the replacement process in detail using the example in Fig. 1. The example uses a small 3-way cache with 8 lines per way. Letters A-Z denote *cache blocks*, and numbers denote *hash values*. Fig. 1g shows the timeline of reads and writes to the tag and data arrays, and the memory bus. Throughout Fig. 1, addresses and hash values obtained in the same access share the same color.

*Walk:* Fig. 1a shows the initial contents of the cache and the miss for address Y that triggers the process. Initially, the addresses returned by the tag lookup for Y are our only replacement candidates for the incoming block (addresses A, D and M). These are the *first-level* candidates. A skew-associative cache would only consider these candidates. In a zcache, the controller starts the walk to expand the number of candidates by computing the hash values of these addresses, shown in Fig. 1b. One of the hash values always matches the hash value of the incoming block. The others denote the positions in the array where we could *move* each of our current replacement candidates to accommodate the incoming block. For example, as column A in Fig. 1b shows, we could move block A to line 2 in way 1 (evicting K) or line 1 in way 2 (evicting X) and write incoming block Y in line 5 of way 0.

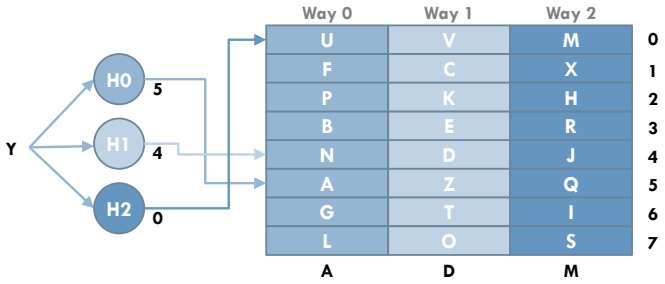
We take the six non-matching hash values in Fig. 1b and perform two accesses, giving us an additional set of six *second-level* replacement candidates, as shown in Fig. 1c (addresses B, K, X, P, Z, and S). We can repeat this process (which, at its core, is a breadth-first graph walk) indefinitely, getting more and more replacement candidates. In practice, we eventually need to stop the walk and select the best candidate found so far. In this example, we expand up to a third level, reaching 21 (3+6+12) replacement candidates. In general, it is not necessary to obtain full levels. Fig. 1d shows a tree with the three levels of candidates. Note how, in expanding the second level, some hash values are repeated and lead to the same address. These *repeats* are bound to happen in this small example, but are very rare in larger caches with hundreds of blocks per way.

*Relocations:* Once the walk finishes, the replacement policy chooses the best replacement candidate. We discuss the implementation of replacement policies in Section III-E. In our example, block N is the best candidate, as shown in Fig. 1d. To accommodate the incoming block Y, the zcache evicts N and relocates its ancestors in the tree (both data and tags), as shown in Fig. 1e. This involves reading and writing the tags and data to their new locations, as the timeline in Fig. 1g indicates. Fig. 1f shows the contents of the cache after the replacement process is finished, with N evicted and Y in the cache. Note how N and Y both used way 0, but completely different locations.

### B. General figures of merit

A zcache with W ways where the walk is limited to L levels has the following figures of merit:

- Replacement candidates ($R$): Assuming no repeats when expanding the tree, $R = W \sum_{l=0}^{L-1} (W-1)^l$.
- Replacement process energy ($E_{miss}$): If the energies to read/write tag or data in a single way are denoted $E_{rt}$, $E_{wt}$, $E_{rd}$ and $E_{wd}$, then $E_{miss} = E_{walk} + E_{relocs} = R \times E_{rt} + m \times (E_{rt} + E_{rd} + E_{wt} + E_{wd})$, where $m \in \{0, .., L-1\}$ is the number of relocations. Note that reads and writes to the data array, which consume most of the energy, grow with $L$, i.e. *logarithmically* with $R$.
- Replacement process latency: Because accesses in a walk can be pipelined, the latency of a walk grows with the number of levels, unless there are so many accesses on each level that they fully cover the latency of a tag array read: $T_{walk} = \sum_{l=0}^{L-1} \max(T_{tag}, (W-1)^l)$. This means that, for $W > 2$, we can get tens of candidates in a small amount of delay. For example, Fig. 1g assumes a tag read delay of 4 cycles, and shows how the walk process for 21 candidates (3 levels) completes in $4 \times 3 = 12$ cycles. The whole process finishes in 20 cycles, much earlier than the 100 cycles used to retrieve the incoming block from main memory.



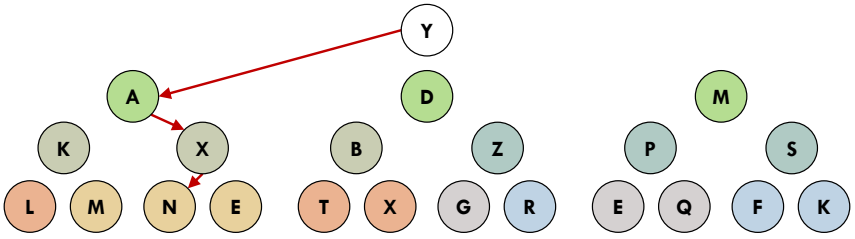

(a) Initial state of the cache and initial miss

| Addr | Y | A | D | M |
|---|---|---|---|---|
| H0 | 5 | 5 | 3 | 2 |
| H1 | 4 | 2 | 4 | 5 |
| H2 | 0 | 1 | 7 | 0 |

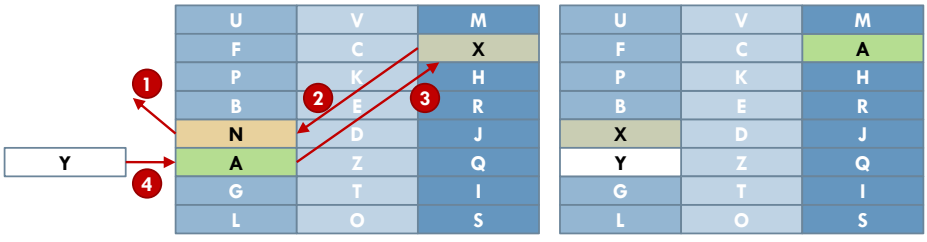
(b) Hash values of first-level candidates

| B | K | X | P | Z | S | Addr |
|---|---|---|---|---|---|---|
| 3 | 7 | 4 | 2 | 6 | 1 | H0 |
| 6 | 2 | 3 | 3 | 5 | 2 | H1 |
| 1 | 0 | 1 | 5 | 3 | 7 | H2 |

(c) Hash values of second-level candidates

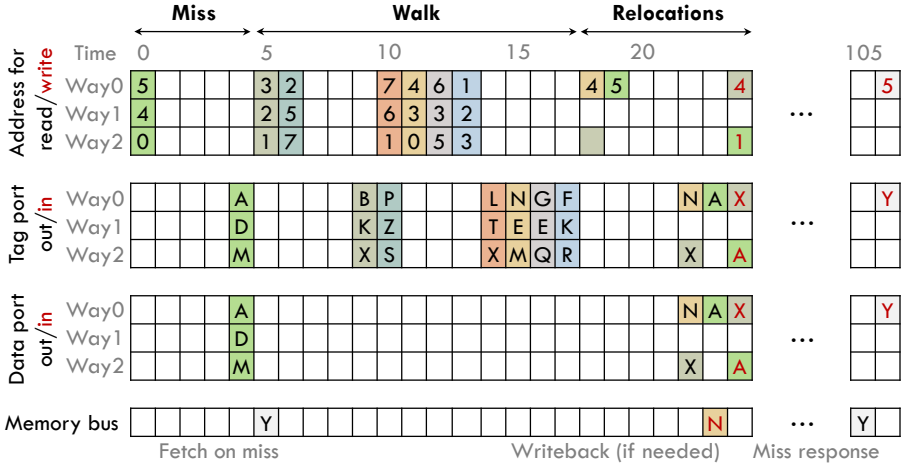



(d) The three levels of replacement candidates. N is selected by the replacement policy




(e) Relocations done to accommodate the incoming block

(f) Final cache state after replacement




(g) Timeline of requests and responses

Fig. 1. Replacement process in a zcache

### C. Implementation

To implement the replacement process, the cache controller needs some modifications involving hash functions, some additional state and, for non-blocking caches, scheduling of concurrent operations.

*Hash functions:* We need one hash function per way. Hash functions range from extremely simple (e.g. bit selection) to exceedingly complex (e.g. cryptographic hash functions like SHA-1). In this study, we use $H_3$ hash functions [11], a family of low-cost, universal, pairwise-independent hash functions that require a few XOR gates per hash bit [38].

*State:* The controller needs to remember the positions of the replacement candidates visited during the walk and the position of the best eviction candidate. Tracking only the most desirable replacement candidate is not sufficient, because relocations need to know about all blocks in the path. However, a single-ported SRAM or small register file suffices. Note that we do not have to remember full tags, just hash values. Also, no back-pointers need to be stored, because for a certain position in the SRAM, the parent's position is always the same. In the example shown in Fig. 1, the controller needs 63 bits of state to track candidates (21 hash values $\times$ 3 bits/value). If the cache was larger, e.g. 3MB, with 1MB per way and 64-byte lines (requiring 14 bits/hash value), it would need 294 bits. Additionally, the controller must buffer the tags and data of the L lines it reads and writes on a relocation. Since the number of levels is typically small (2 or 3 in our experiments), this also entails a small overhead.

*Concurrent operations for non-blocking caches:* To avoid increasing cache latency, the replacement process should be able to run concurrently with all other operations (tag/data reads and writes due to hits, write-backs, invalidations, etc.). The walk process can run concurrently without interference. This may lead to benign races where, for example, the walk identifies the best eviction candidate to be a block that was accessed (e.g. with a hit) in the interim. This is exceedingly rare in large caches, so we simply evict the block anyway. In smaller caches (e.g. highly-associative but small TLBs or first-level caches), we could keep track of the best two or three eviction candidates and discard them if they are accessed while the walk process is running.

In the second part of the replacement, the relocations, the controller must block intervening operations to at most L positions while blocks in these positions are being relocated. We note that the controller already has logic to deal with these cases (e.g. with MSHRs [28]).

While it is feasible to run multiple replacement processes concurrently, it would complicate the cache controller, and since replacements are not in the critical path, they can simply queue up. Concurrent replacements would only make sense to increase bandwidth utilization when the cache is close to bandwidth saturation. As shown in Section VI, we do not see the need for such mechanism in our experiments.

In conclusion, the zcache imposes minor state and logic overheads to traditional cache controllers.

### D. Extensions

We now discuss additional implementation options to enhance zcaches.

*Avoiding repeats:* In small first-level caches or TLBs, repeats can be common due to walking a significant portion of the cache. Moreover, a repeat at a low level can trigger the expansion of many repeated candidates. Repeats can be avoided by inserting the addresses visited during the walk in a Bloom filter [6], and not continuing the walk through addresses that are already represented in the filter. Repeats are rare in our experiments, so we do not see any performance benefit from this.

*Alternative walk strategies:* The current walk performs a breadth-first search for candidates, fully expanding all levels. Alternatively, we could perform a depth-first search (DFS), always moving towards higher levels of replacement candidates. Cuckoo hashing [35] follows this strategy. DFS allows us to remove the walk table and interleave walk with relocations, reducing state. However, it increases the number of relocations for a given number of replacement candidates (since $L = R/W$), which in turn increases both the energy required per replacement (as relocations read and write to the much wider data array) and replacement latency (as accesses in the walk cannot be pipelined). BFS is a much better match to a hardware implementation as the extra required state for BFS is a few hundred bits at most. Nevertheless, a controller can implement a hybrid BFS+DFS strategy to increase associativity cheaply. For instance, in our example in Fig. 1, the controller could perform a second phase of BFS, trying to re-insert N rather than evicting it, to double the number of candidates without increasing the state needed.

### E. Replacement Policy

So far, we have purposely ignored how the replacement policy is implemented. In this section, we cover how to implement or approximate LRU. While set-associative caches can cheaply maintain an order of the blocks in each set (e.g. using LRU or pseudo-LRU), since the concept of a set does not exist in a zcache, policies that rely on this ordering need to be implemented differently. However, several processor designs already find it too expensive to implement set ordering and resort to policies that do not require it [20, 41]. Additionally, some of the latest, highest-performing policies do not rely on set ordering [24]. While designing a replacement policy specifically tailored to zcaches is an interesting endeavor, we defer it to future work.

*Full LRU:* We use a global timestamp counter, and add a timestamp field to each block in the cache. On each access, the timestamp counter is incremented, and the timestamp field is updated to the current counter value. On a replacement, the controller selects the replacement candidate with the lowest timestamp (in $\mod 2^n$ arithmetic). This design requires very simple logic, but timestamps have to be large (e.g. 32 bits) to make wrap-arounds rare, thus having high area overhead.

*Bucketed LRU:* To decrease space overheads, timestamps are made smaller, and the controller increases the timestamp counter once every $k$ accesses. For example, with $k = 5\%$ of the cache size and $n = 8$ bits per timestamp, it is rare for a block to survive a wrap-around without being either accessed or evicted. We use this LRU policy in our evaluation.

## IV. An Analytical Framework for Associativity

Quantifying and comparing associativity across different cache designs is hard. In set-associative caches, more ways implicitly mean higher associativity. However, when comparing different designs (e.g. a set-associative cache and a zcache), the number of ways becomes a useless proxy for associativity.

The most commonly used approach to quantify associativity is by the number of conflict misses [21]. Conflict misses for a cache are calculated by subtracting the number of misses incurred by a fully-associative cache of the same size from the total number of misses. Using conflict misses as a proxy for associativity has the advantage of being an end-to-end metric, directly linking associativity to performance. However, it is subject to three problems. First, it is highly dependent on the replacement policy; for example, by using an LRU replacement policy in a workload with an anti-LRU access pattern, we can get higher conflict misses when increasing the number of ways. Second, in CMPs with multilevel memory hierarchies, changing the associativity can alter the reference stream at higher cache levels, and comparing the number of conflict misses when the total number of accesses differs is meaningless. Finally, conflict misses are workload-dependent, so they cannot be used as a general proxy for associativity.

In this section, we develop a framework to address these issues, with the objectives of 1) being able to compare associativity between different cache organizations, and 2) determining how various design aspects (e.g. ways, number of replacement candidates, etc) influence cache associativity.

### A. Associativity Distribution

*Model:* We divide a cache into the following components:

- Cache array: Holds tags and data, implements associative lookups by block address, and, on a replacement, gives a list of replacement candidates that can be evicted.
- Replacement policy: Maintains a *global* rank of which cache blocks to replace.

This model assumes very little about the underlying cache implementation. The array could be set-associative, a zcache, or any of the schemes mentioned in Section II. The only requirement that we impose on the replacement policy is to define a global ordering of blocks, which most policies inherently do. For example, in LRU blocks are ranked by the time of their last reference, in LFU they are ordered by access frequency, and in OPT [4] they are ranked by the time to their next reference. This does not mean that the implementation actually maintains this global rank. In a set-associative cache, LRU only needs to remember the order of elements in each set, and in a zcache this can be achieved with timestamps, as explained in Section III-E.

By convention, blocks with a higher preference to be evicted are given a higher rank $r$. In a cache with $B$ blocks, $r \in [0, ..., B-1]$. To make the rest of the analysis independent of cache size, we define a block's *eviction priority* to be its rank normalized to $[0, 1]$, i.e. $e = r/(B-1)$.

*Associativity distribution:* We define the associativity distribution as the *probability distribution* of the eviction priorities of evicted blocks. In a fully-associative cache, we would always evict the block with $e = 1.0$. However, most cache designs examine only a small subset of the blocks in an eviction, so they select blocks with lower eviction priorities. In general, the more skewed the distribution is towards $e = 1.0$, the higher the associativity is. The associativity distribution characterizes the *quality* of the replacement decisions made by the cache in a way that is independent of the replacement policy. Note that this decouples how the array performs from ill-effects from the replacement policy. For example, a highly associative cache may always find replacement candidates with high eviction priorities, but if the replacement policy does a poor job in ranking the blocks, this may actually hurt performance.

### B. Linking Associativity and Replacement Candidates

Defining associativity as a probability distribution lets us evaluate the quality of the replacement candidates, but is still dependent on workload and replacement policy. However, under certain general conditions this distribution can be characterized by a single number, the number of replacement candidates. This is the figure of merit optimized by zcaches.

*Uniformity assumption:* If the cache array always returns $n$ replacement candidates, and we treat the eviction priorities of these blocks as random variables $E_i$, assuming that they are 1) *uniformly distributed* in [0,1] and 2) statistically *independent* from each other, we can derive the associativity distribution. Since $E_1, ..., E_n \sim U[0, 1], i.i.d$, the cumulative distribution function (CDF) of each eviction priority is $F_{E_i}(x) = Prob(E_i \leq x) = x, x \in [0, 1]$[1]. The associativity is the random variable $A = \max\{E_1, ..., E_n\}$, and its CDF is:

$$\begin{aligned} F_A(x) &= Prob(A \leq x) = Prob(E_1 \leq x \wedge ... \wedge E_n \leq x) \\ &= Prob(E_i \leq x)^n = x^n, x \in [0, 1] \end{aligned}$$

Therefore, under this *uniformity assumption*, the associativity distribution only depends on $n$, the number of replacement candidates. Fig. 2 shows example CDFs of the associativity distribution, in linear and semi-log scales, with each line representing a different number of replacement candidates. The higher the number of replacement candidates, the more skewed towards 1.0 the associativity distribution becomes. Also, evictions of blocks with a low eviction priority quickly become very rare. For example, for 16 replacement candidates, the probability of evicting a block with $e < 0.4$ is $10^{-6}$.

[1] Note that we are treating $E_i$ as *continuous* random variables, even though they are *discrete* (normalized ranks with one of $B$ equally probable values in $[0, 1]$). We do this to achieve results that are independent of cache size $B$. Results are the same for the discretized version of these equations.

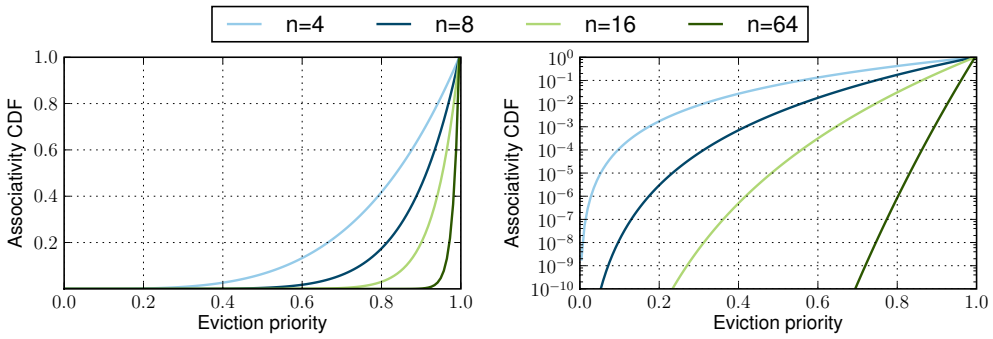
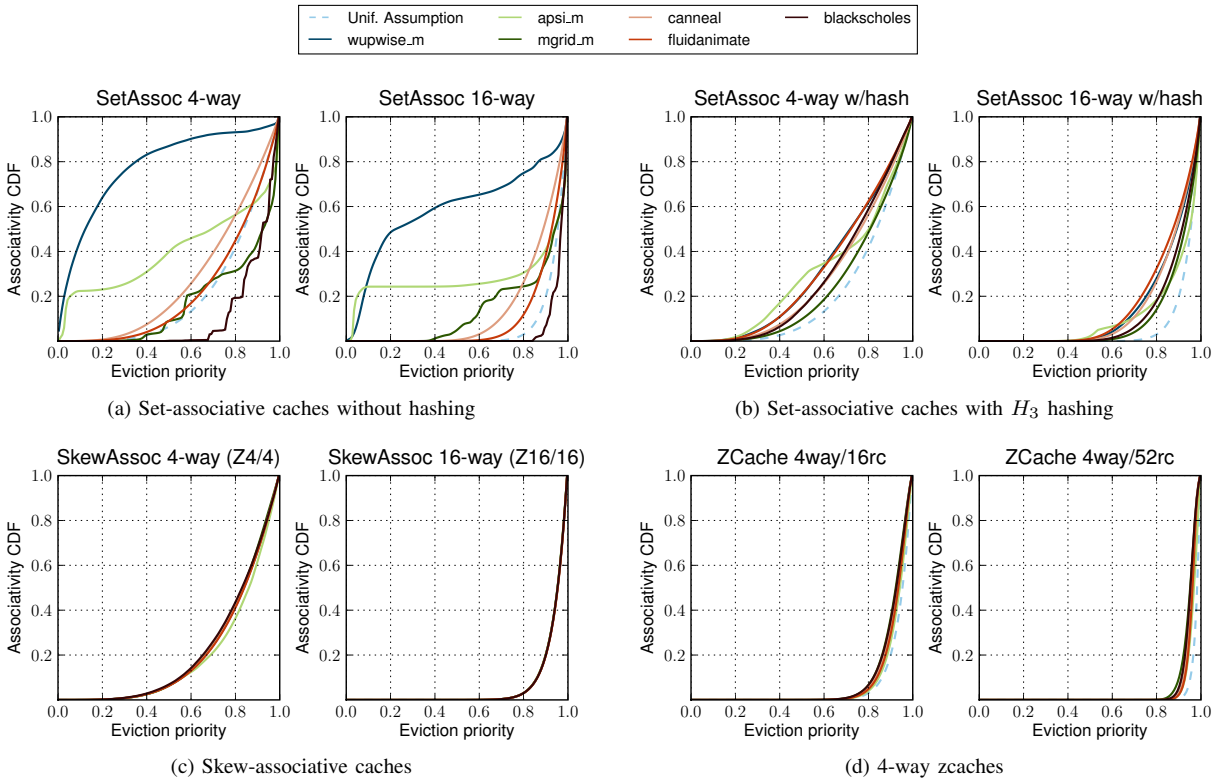



Fig. 2. Associativity CDFs under the uniformity assumption ($F_A(x) = x^n, x \in [0, 1]$) for $n = 4, 8, 16, 64$ candidates, in linear and logarithmic scales.




(a) Set-associative caches without hashing

(b) Set-associative caches with $H_3$ hashing

(c) Skew-associative caches

(d) 4-way zcaches

Fig. 3. Associativity distributions for selected PARSEC and SPECOMP workloads using different types of caches.

*Random candidates cache:* The uniformity assumption makes it simple to characterize associativity, but it is not met in general by real cache designs. However, a cache array that returns $n$ randomly selected replacement candidates (with repetition) from all the blocks in the cache always achieves these associativity curves *perfectly*. Each $E_i$ is uniformly distributed because it is an unbiased *random sampling* of one of the $B$ possible values of a rank, and since different selections are done independently, the $E_i$ are independent as well. We simulated this cache design with tens of real workloads, under several configurations and replacement policies, and obtained associativity distributions as shown in Fig. 2, experimentally validating the previous derivation.

Although this *random candidates* cache design is unrealistic, it reveals a sufficient condition to achieve the uniformity assumption: the more randomized the replacement candidates, the better a cache will match the uniformity assumption.

### C. Associativity Measurements of Real Caches

Our analytical framework implies that the number of replacement candidates is the key metric in determining associativity. We now evaluate whether this is the case using real cache designs.

*Set-associative caches:* Fig. 3a shows the associativity distributions for 8MB L2 set-associative caches of 4 and 16 ways, using an LRU replacement policy. The details on system configuration and methodology can be found in Section V. Each of the 6 solid lines represents a different benchmark,

| | |
|---|---|
| **Cores** | 32 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz |
| **L1 caches** | 32 KB, 4-way set associative, split D/I, 1-cycle latency |
| **L2 cache** | 8 MB NUCA , 8 banks, 1 MB bank, shared, inclusive, MESI directory coherence, 4-cycle average L1-to-L2-bank latency, 6–11-cycle L2 bank latency |
| **MCU** | 4 memory controllers, 200 cycles zero-load latency, 64 GB/s peak memory BW |

TABLE I
MAIN CHARACTERISTICS OF THE SIMULATED CMPS. THE LATENCIES ASSUME A 32 NM PROCESS AT 2GHZ.

from a representative selection of PARSEC and SPECOMP applications. The single dotted line per graph plots the associativity distribution under the uniformity assumption, which is independent of the workload. We see that the distributions differ significantly from the uniformity assumption. Two workloads (wupwise and apsi) do significantly worse, with the CDF rapidly climbing towards 1.0. For example, in wupwise, 60% of the evictions happen to blocks with $\leq$ 20% eviction priority. Others (mgrid, canneal and fluidanimate) have sensibly worse associativity, and only one benchmark (blackscholes) outperforms the uniformity assumption. These differences are not surprising: replacement candidates all come from the same small set, thwarting independence, and locality of reference will skew eviction priorities towards lower values, breaking the assumption of an uniform distribution.

We can improve associativity with hashing. Fig. 3b shows the associativity distributions of set-associative caches indexed by an $H_3$ hash of the block address. Associativity distributions generally improve, but some hot-spots remain, and all workloads now perform sensibly worse than the uniformity assumption case.

*Skew-associative caches and zcaches:* Fig. 3c shows the associativity distributions of 4 and 16-way skew-associative caches. As we can see, skew-associative caches closely match the uniformity assumption on all workloads. These results provide an analytical foundation to the previous empirical observations that skew-associative caches "improve performance predictability" [7].

Fig. 3d shows the associativity of 4-way zcaches with 2 and 3 levels of replacement candidates. We also observe a close match to the uniformity assumption. This is expected, since replacement candidates are even more randomized: $n^{th}$-level candidates depend on the addresses of the $(n-1)^{th}$-level candidates, making the set of positions checked varying with cache contents.

In conclusion, both skew-associative caches and zcaches match the uniformity assumption in practice. Hence, their associativity is directly linked to the number of candidates examined on replacement. Although the graphs only show a small set of applications for clarity, results with other workloads and replacement policies are essentially identical. The small differences observed between applications decrease by either increasing the number of ways (and hash functions) or improving the quality of hash functions (the same experiments using more complex SHA-1 hash functions instead of $H_3$ yield distributions identical to the uniformity assumption).

Overall, our analysis framework reveals two main results:

1) In a zcache, associativity is determined by the *number of replacement candidates*, and not the number of ways, essentially *decoupling ways and associativity*.
2) When using an equal number of replacement candidates, zcaches empirically show better associativity than set-associative caches for most applications.

## V. EXPERIMENTAL METHODOLOGY

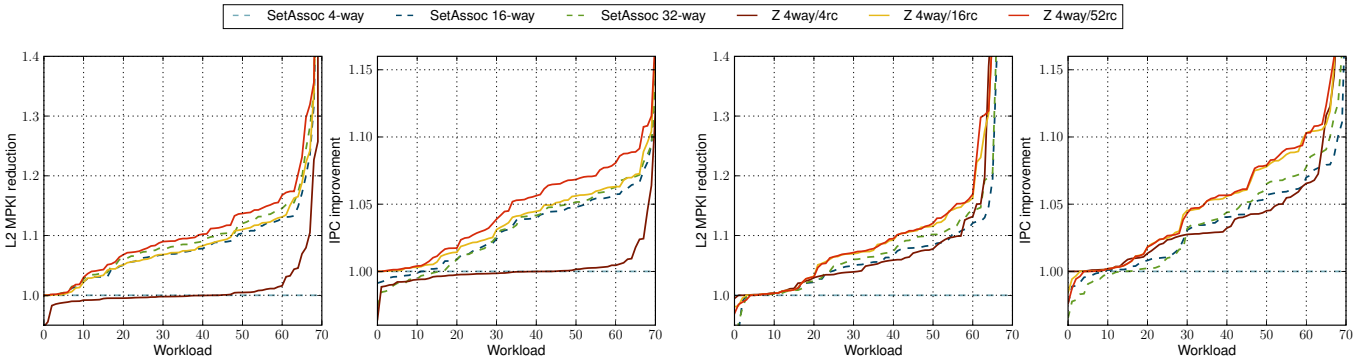
*Infrastructure:* We perform microarchitectural, execution-driven simulation using an x86-64 simulator based on Pin [31]. We use McPAT [30] to obtain comprehensive timing, area and energy estimations for the CMPs we model, and use CACTI 6.5 [33] for more detailed cache area, power and timing models. We use 32nm ITRS models, with a high-performance process for all the components of the chip except the L2 cache, which uses a low-leakage process.

*System:* We model a 32-core CMP, with in-order x86 cores modeled after the Atom processor [17]. The system has a 2-level cache hierarchy, with a fully shared L2 cache. Table I shows the details of the system. On 32nm, this CMP requires about $220mm^2$ and has a TDP of around 90W at 2GHz, both reasonable budgets.

*Workloads:* We use a variety of multithreaded and multiprogrammed benchmarks: 6 PARSEC [5] applications (blackscholes, canneal, fluidanimate, freqmine, streamcluster and swaptions), 10 SPECOMP benchmarks (all except galgel, which gcc cannot compile) and 26 SPECCPU2006 programs (all except dealII, tonto and wrf, which we could not compile). For multiprogrammed runs, we run different instances of the same single-threaded CPU2006 application on each core, plus 30 random CPU2006 workload combinations (choosing 32 workloads each time, with repetitions allowed). These make a total of 72 workloads. All applications are run with their reference (maximum size) input sets. For multithreaded workloads, we fast-forward into the parallel region and run the first 10 billion instructions. Since synchronization can skew IPC results for multithreaded workloads [2], we do not count instructions in synchronization routines (locks, barriers, etc.) to determine when to stop execution, but we do include them in energy calculations. For multiprogrammed workloads, we follow standard methodology from prior work [24]: we fast-forward 20 billion instructions for each process, simulate until all the threads have executed at least 256 million instructions, and only take the first 256 million instructions of each thread into account for IPC computations.

| Cache Type | Serial lookups | | | Parallel lookups | | | | |
|---|---|---|---|---|---|---|---|---|
| | Bank latency | Bank E/hit | Bank E/miss | Bank latency | Bank E/hit | Bank E/miss | L2 area | L2 leakage |
| **SetAssoc 4-way** | 4.14 ns | 0.61 nJ | 1.26 nJ | 2.91 ns | 0.71 nJ | 1.42 nJ | 42.3 $mm^2$ | 535 mW |
| **SetAssoc 8-way** | 4.41 ns | 0.75 nJ | 1.57 nJ | 3.18 ns | 0.99 nJ | 1.88 nJ | 45.1 $mm^2$ | 536 mW |
| **SetAssoc 16-way** | 4.74 ns | 0.88 nJ | 1.87 nJ | 3.51 ns | 1.42 nJ | 2.46 nJ | 46.4 $mm^2$ | 561 mW |
| **SetAssoc 32-way** | 5.05 ns | 1.23 nJ | 2.66 nJ | 3.82 ns | 2.34 nJ | 3.82 nJ | 51.9 $mm^2$ | 588 mW |
| **ZCache 4/16** | 4.14 ns | 0.62 nJ | 2.28 nJ | 2.91 ns | 0.72 nJ | 2.44 nJ | 42.3 $mm^2$ | 535 mW |
| **ZCache 4/52** | 4.14 ns | 0.62 nJ | 3.47 nJ | 2.91 ns | 0.72 nJ | 3.63 nJ | 42.3 $mm^2$ | 535 mW |

TABLE II
AREA, POWER AND LATENCY OF 8MB, 8-BANKED L2 CACHES WITH DIFFERENT ORGANIZATIONS.




(a) OPT replacement

(b) LRU replacement

Fig. 4. L2 MPKI and IPC improvements for all workloads, over a 4-way set-associative with hashing baseline.

## VI. EVALUATION OF ZCACHE AS A LAST-LEVEL CACHE

The zcache can be used with any design that requires high associativity at low overheads in terms of area, hit time, and hit energy. In this paper, we evaluate zcache as a last-level cache in a 32-node CMP. We defer other use cases, such as first-level caches or TLBs, to future work. We first quantify the area, energy and latency advantages of zcaches versus set-associative caches with similar associativity, then compare the performance and system-wide energy over our set of workloads.

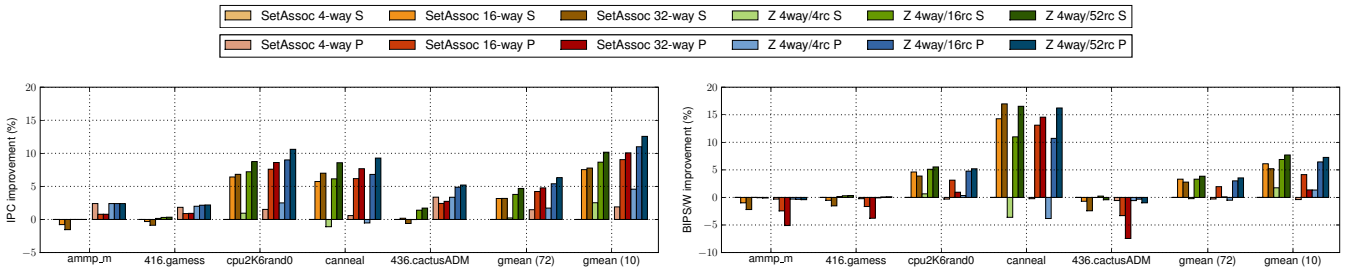
### A. Cache Costs

Table II shows the timing, area and power requirements of both set-associative caches and zcaches with varying associativities. We use CACTI's models to obtain these numbers. Tag and data arrays are designed separately by doing a full design space exploration and choosing the design that minimizes area×delay×power. Arrays are sub-banked, and both the address and data ports are implemented using H-trees. We show results for both serial and parallel-lookup caches. In serial caches, tag and data arrays are accessed sequentially, saving energy at the expense of delay. In parallel caches, both tag and data accesses are initiated in parallel. When the tag read resolves the appropriate way, it propagates a way-select signal to the data array, which selects and propagates the correct output. This parallelizes most of the tag and data accesses while avoiding an exceedingly wide data array port. For zcaches, we explore designs with two and three-level walks. We denote zcaches with "$W/R$", indicating the number of ways and replacement candidates, respectively. For example, a 4/16 zcache has 4 ways and 16 replacement candidates per eviction (obtained from a two-level walk).

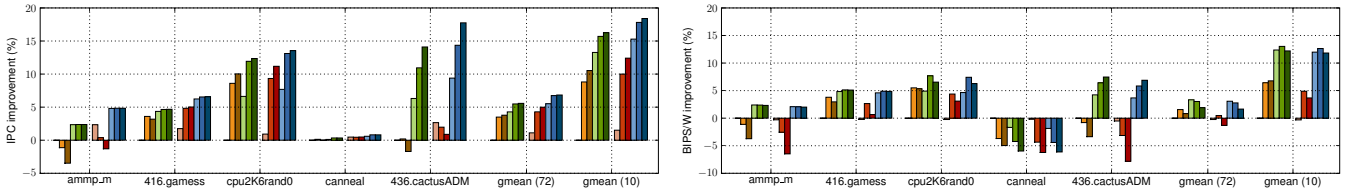
Table II shows that increasing the number of ways beyond 8 starts imposing significant area, latency and energy overheads. For example, a 32-way cache with serial lookups has 1.22× the area, 1.23× the hit latency and 2× the hit energy of a 4-way cache (for parallel lookups, hit latency is 1.32× and hit energy is 3.3×). This is logical, since a 32-way cache reads 4× more tag bits than data bits per lookup, the tag array has a much wider port, and the critical path is longer (slower tag array, more comparators). For zcaches, however, area, hit latency and hit energy grow with the number of ways, but not with the number of replacement candidates. This comes at the expense of increasing energy per miss, which, however, is still similar to set-associative caches with the same associativity. For example, a serial-lookup zcache 4/52 has almost twice the associativity of a 32-way set-associative cache at 1.3× higher energy per miss, but retains the 2× lower hit energy and 1.23× lower access latency of a 4-way cache.

### B. Performance

Fig. 4 shows the improvements in both L2 misses per thousand instructions (MPKI) and IPC for all workloads, using both OPT and LRU replacement policies. Each line represents the improvement of a different cache design over a baseline




(a) OPT replacement




(b) LRU replacement

Fig. 5. IPC and energy efficiency (BIPS/W) improvements for serial and parallel-lookup caches, over a serial-lookup 4-way set-associative with hashing baseline. Each graph shows improvements for 5 representative workloads, plus the geometric mean over both all 72 workloads and the 10 workloads with the highest L2 MPKI.

4-way set-associative cache with $H_3$ hashing. Caches without hashing perform significantly worse (even at high associativities), so we do not consider them here. Serial-lookup caches are used in all cases. For each line, workloads (in the x-axis) are sorted according to the improvement achieved, so each line is monotonically increasing. Fractional improvements are given (e.g. a L2 MPKI reduction of 1.2 means 1.2× lower MPKI than the baseline).

*OPT:* Fig. 4a shows the effects of using OPT replacement (i.e. evicting the candidate reused furthest). OPT simulations are run in trace-driven mode. Although OPT is unrealistic, it removes ill-effects from the replacement policy (where e.g. increasing associativity degrades performance), allowing us to decouple replacement policy issues from associativity effects[2]. Note that these numbers do not necessarily show maximum improvements from increasing associativity, as other replacement policies may be more sensitive to associativity changes. In terms of misses, higher associativities always improve MPKI, and designs with the same associativity have practically the same improvements (e.g. 16-way set-associative vs Z4/16). However, for set-associative caches, these improvements in MPKI do not always translate to IPC, due to the additional access latency (1 extra cycle for 16-way, 2 cycles for 32-way). For example, the 32-way set-associative design performs worse than the 4-way design on 15 workloads (which have a large number of L1 misses, but few L2 misses), and performs worse than the 16-way design on half of the workloads (36). In contrast, zcaches do not suffer from increased access latency, sensibly improving IPC with associativity for all workloads (e.g. a Z4/52 improves IPC by up to 16% over the baseline).

[2]In caches with interference across sets, like skew-associative and zcaches, OPT is not actually optimal, but it is a good heuristic.

*LRU:* Fig. 4b compares cache designs when using LRU. Associativity improves MPKI for all but 3 workloads, and both MPKI and IPC improvements are significant (e.g. a Z4/52 reduces L2 misses by up to 2.1× and improves performance by up to 25% over a 4-way set-associative cache). With LRU, the difference between Z4/16 and Z4/52 designs is lower than with OPT, however they significantly outperform both the baseline and the Z4/4 (skew-associative) design.

### C. Serial vs Parallel-Lookup Caches

Fig. 5 shows the performance and *system-wide* energy efficiency when using serial and parallel-lookup caches, under both OPT and LRU replacement policies. Results are normalized to a serial-lookup, 4-way set-associative cache with $H_3$ hashing. Each graph shows improvements on five representative applications, as well as the geometric means of both all 72 workloads and the 10 workloads with the highest L2 MPKI.

We can distinguish three types of applications: a few benchmarks, like blackscholes or freqmine, have low L1 miss rates, and are insensitive to the L2's organization. Other applications, like ammp and gamess, have frequent L2 hits but infrequent L2 misses. These workloads are sensitive to hit latency, so parallel-lookup caches provide higher performance gains than increasing associativity (e.g. a 3% IPC improvement on gamess vs serial-lookup caches). In fact, increasing associativity in set-associative caches reduces performance due to higher hit latencies, while highly-associative zcaches do not degrade performance. Finally, workloads like cpu2K6rand0, canneal, and cactusADM have frequent L2 misses. These applications are often sensitive to associativity, and a highly-associative cache improves performance (by reducing L2 MPKI) more than reducing access time (e.g. in cactusADM with LRU,

going from Z4/4 to Z4/52 improves IPC by 9%, while going from serial to parallel-lookup improves IPC by 3%).

In terms of energy efficiency, set-associative caches and zcaches show different behaviors when increasing associativity. Because hit energy increases steeply with the number of ways in parallel-lookup caches, 16 and 32-way set-associative caches often achieve lower energy efficiency than serial-lookup caches (e.g. up to 8% lower BIPS/W in cactusADM). In contrast, serial and parallel-lookup zcaches achieve practically the same energy efficiency on most workloads, due to their similarly low access and miss energies. In conclusion, zcaches enable highly-associative, energy-efficient parallel-lookup caches.

Overall, zcaches offer both the best performance and energy efficiency. For example, under LRU, when considering all 72 workloads, a parallel-lookup zcache 4/52 improves IPC by 7% and BIPS/W by 3% over the 4-way baseline. Over the subset of the 10 most L2 miss-intensive workloads, a zcache 4/52 improves IPC by 18% and energy efficiency by 13% over the 4-way baseline, and obtains 7% higher performance and 10% better energy efficiency than a 32-way set-associative cache.

### D. Array Bandwidth

Since zcaches perform multiple tag lookups on a miss, it is worth examining whether these additional lookups can saturate bandwidth. Of the 72 workloads, the maximum average load per bank is 15.2% (i.e. 0.152 core accesses/cycle/L2 bank). However, as L2 misses increase, average load decreases: at 0.005 misses/cycle/bank, average load is 0.035 accesses/cycle/bank, and total load on the tag array for a Z4/52 cache is 0.092 tag accesses/cycle/bank. In other words, as L2 misses increase, bandwidth pressure on the L2 decreases; the system is self-throttling. ZCaches use this spare tag bandwidth to improve associativity. Ultimately, even for high-MLP architectures, the load on the tag arrays is limited by main memory bandwidth, which is more than an order of magnitude smaller than the maximum L2 tag bandwidth and much harder to scale.

## VII. Related Work

The zcache is inspired by cuckoo hashing, a technique to build space-efficient hash tables proposed by Pagh and Rodler [35]. The original design uses two hash functions to index the hash table, so each lookup needs to check two locations. On an insertion, if both possible locations are occupied, the incoming item replaces one of them at random, and the replaced block is reinserted. This is repeated until either an empty location is found or, if a limit number of retries is reached, elements are rehashed into a larger array. Though cuckoo hashing has been mostly studied as a technique for software hash tables, hardware variants have been proposed to implement lookup tables in IP routers [16]. For additional references, Mitzenmacher has a survey on recent research in cuckoo hashing [32].

Both high associativity and a good replacement policy are necessary to improve cache performance. The growing importance of cache performance has sparked research into alternative policies that outperform LRU [14, 23, 24, 44]. The increasing importance of on-chip wire delay has also motivated research in non-uniform cache architectures (NUCA) [27]. Some NUCA designs such as NuRAPID [15] use indirection to enhance the flexibility of NUCA placement and reduce access latency instead of increasing associativity.

## VIII. Conclusions and Future Work

We have presented the zcache, a cache design that enables high associativity with a small number of ways. The zcache uses a different hash function per way to enable an arbitrarily large number of replacement candidates on a miss. To evaluate the zcache's associativity, we have developed a novel analytical framework to characterize and compare associativity. We use this framework to show that, for zcaches, associativity is determined by the number of replacement candidates, not the number of ways, hence decoupling ways and associativity. An evaluation using zcaches as the last-level cache in a CMP shows that they provide high associativity with low overheads in terms of area, hit time, and hit energy. ZCaches outperform traditional set-associative caches in both performance and energy efficiency, with a 4-way zcache achieving both 18% higher performance and 13% higher performance/watt than 4-way set-associative counterpart over a set of 10 L2 miss-intensive workloads, and 7% higher performance and 10% better energy efficiency than a 32-way set-associative cache.

There are several opportunities for further research, such as using zcaches to build highly associative first-level caches and TLBs for multithreaded cores. Additionally, replacement policies that are specifically suited to the zcache could be designed. Finally, since the zcache makes it trivial to increase or reduce associativity with the same hardware design, it would be interesting to explore adaptive replacement schemes that use the high associativity only when it improves performance, saving cache bandwidth and energy when high associativity is not needed, or even making associativity a software-controlled property.

## Acknowledgements

We sincerely thank John Brunhaver, Christina Delimitrou, David Lo, George Michelogiannakis and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. Daniel Sanchez was supported by a Hewlett-Packard Stanford School of Engineering Fellowship.

## References

[1] A. Agarwal and S. D. Pudar, "Column-associative caches: a technique for reducing the miss rate of direct-mapped caches," in *Proc. of the 20th annual Intl. Symp. on Computer architecture*, 1993.

[2] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[3] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A new last level cache architecture with global block priority," in *Proc. of the 40th annual IEEE/ACM Intl Symp. on Microarchitecture*, 2007.

[4] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, 1966.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, 1970.

[7] F. Bodin and A. Seznec, "Skewed associativity enhances performance predictability," in *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995.

[8] A. Bracy, K. Doshi, and Q. Jacobson, "Disintermediated active communication," *Comput. Archit. Lett.*, vol. 5, no. 2, 2006.

[9] M. W. Brehob, "On the mathematics of caching," Ph.D. dissertation, Michigan State University, 2003.

[10] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture*, 1996.

[11] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. of the 9th annual ACM Symposium on Theory of Computing*, 1977.

[12] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency," in *Proc. of the 34th annual Intl. Symp. on Computer architecture*, 2007.

[13] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006.

[14] M. Chaudhuri, "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches," in *Proc. of the 42nd annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2009.

[15] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *Proc. of the 36th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2003.

[16] S. Demetriades, M. Hanna, S. Cho, and R. Melhem, "An efficient hardware-based multi-hash scheme for high speed IP lookup," in *Proc. of the 16th IEEE Symp. on High Performance Interconnects*, 2008.

[17] G. Gerosa *et al.*, "A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile PCs in 45nm hi-K metal gate CMOS," in *IEEE Intl. Solid-State Circuits Conf.*, 2008.

[18] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[19] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proc. of the 31st annual Intl. Symp. on Computer Architecture*, 2004.

[20] Hewlett-Packard, "Inside the Intel Itanium 2 processor," Tech. Rep., 2002.

[21] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Trans. Comput.*, vol. 38, no. 12, 1989.

[22] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[23] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008.

[24] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010.

[25] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. of the 17th annual Intl. Symp. on Computer Architecture*, 1990.

[26] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Proc. of the 10th Intl. Symp. on High Performance Computer Architecture*, 2004.

[27] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[28] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. of the 8th annual Intl. Symp. on Computer Architecture*, 1981.

[29] N. Kurd *et al.*, "Westmere: A family of 32nm IA processors," in *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[30] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2009.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2005.

[32] M. Mitzenmacher, "Some open questions related to cuckoo hashing," in *Proc. of the 17th annual European Symp. on Algorithms*, 2009.

[33] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. of the 40th annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2007.

[34] V. Nagarajan and R. Gupta, "ECMon: exposing cache events for monitoring," in *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.

[35] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Proc. of the 9th annual European Symp. on Algorithms*, 2001.

[36] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *Proc. of the 32nd annual Intl. Symp. on Computer Architecture*, 2005.

[37] D. Rolán, B. B. Fraguela, and R. Doallo, "Adaptive line placement with the set balancing cache," in *Proc. of the 42nd annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2009.

[38] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Proc. of the 40th annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2007.

[39] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[40] J. Shin *et al.*, "A 40nm 16-core 128-thread CMT SPARC SoC processor," in *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[41] Sun Microsystems, "UltraSPARC T2 supplement to the UltraSPARC architecture 2007," Tech. Rep., 2007.

[42] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic, "The bulk multicore architecture for improved programmability," *Commun. ACM*, vol. 52, no. 12, 2009.

[43] D. Wendel *et al.*, "The implementation of POWER7: A highly parallel and scalable multi-core high-end server processor," in *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[44] Y. Xie and G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.

[45] C. Zhang, X. Zhang, and Y. Yan, "Two fast and high-associativity cache schemes," *IEEE Micro*, vol. 17, no. 5, 1997.