Limited Associativity Makes Concurrent Software Caches a Breeze

Dolev Adas
Computer Science
Technion
sdolevfe@cs.technion.ac.il

Gil Einziger
Computer Science
Ben-Gurion University of the Negev
gilein@bgu.ac.il

Roy Friedman
Computer Science
Technion
roy@cs.technion.ac.il

September 8, 2021

Abstract

Software caches optimize the performance of diverse storage systems, databases and other software systems. Existing works on software caches automatically resort to fully associative cache designs. Our work shows that limited associativity caches are a promising direction for concurrent software caches. Specifically, we demonstrate that limited associativity enables simple yet efficient realizations of multiple cache management schemes that can be trivially parallelized. We show that the obtained hit ratio is usually similar to fully associative caches of the same management policy, but the throughput is improved by up to x5 compared to production-grade caching libraries, especially in multi-threaded executions.

1 Introduction

Caching is a fundamental design pattern for boosting systems' performance. Caches store part of the data in a fast and close memory to the program's execution. Accessing data from such a memory, called a *cache*, reduces waiting times, thereby improving running times. A *cache hit* means that an accessed data item is already located in the cache; otherwise, it is a *cache miss*. Usually, only a small fraction of the data can fit inside a given cache. Hence, a *cache management scheme* decides which data items should be placed in the cache to maximize the *cache hit ratio*, i.e., the ratio of cache hits to all accesses. More so, such a management scheme should be lightweight; otherwise, its costs would outweigh its benefits.

At coarse granularity, we can distinguish between *hardware caches* and *software caches*. A CPU's hardware cache uses fast SRAM memory as a cache for the slower main memory, which is made of DRAM [22]. In addition, the main memory (DRAM) is often utilized as a cache for secondary storage (disks and SSDs) or network/remote storage. Similarly, most storage systems include a software-managed cache, e.g., the Velocix media platform stores video chunks in a multi-layer cache [8]. Additional popular examples include storage systems like Redis [7], Cassandra [2], HBase [3] and Accumulo [1], graph databases like DGraph [5] and neo4j [6], etc. Web caching is another popular example.

An important aspect of the cache management scheme is to select a *victim* for eviction whenever there is not enough room in the cache to admit a newly accessed data item. In *fully associative* caches, the cache management schemes can evict any of the cached items, as illustrated in Figure 1. In contrast, *limited associativity* caches restrict the selection of data items that may be evicted. Specifically, limited associativity caches are portioned into independent *sets*. Each set consists of k *ways* (places), each of which can store a data item. When admitting an item to a limited associativity cache, one must select the cache victim from the same set as the newly admitted item. In such caches, known as *k-way set associative caches*, items' IDs are mapped into sets using a hash function and each set is an *independent* sub-cache of size k.

(a) Fully Associative Cache

(b) Limited Associativity Cache

Figure 1: A schematic overview of a fully associative cache (a) and a limited associativity cache (b). In a fully associative cache, Alice and Bob operate on a shared resource and require some form of synchronization between them. In the limited associativity case, the cache is partitioned into many independent sub-caches, also known as sets, and updates are assigned to sets through hashing. Most times, Alice and Bob operate on different sets requiring no synchronization at all.

In hardware, limited associativity caches are simpler to implement, consume less power to operate and are often significantly faster than fully associative designs. Thus, the vast majority of hardware-based caches have limited associativity. In contrast, non-distributed software-based caches are almost always fully associative. Specifically, cache implementations usually employ hash tables as building blocks. This is often so obvious that papers describing software caches do not even explicitly mention that their algorithms are fully associative [14, 17, 33, 41]. A possible reason for this lack of previous consideration stems from the understanding that fully associative caches should yield higher hit ratios than the same sized limited associativity caches. Further, it is relatively easy to implement such caches from fully associative building blocks such as concurrent hash tables.

However, our work argues that software caches significantly benefit from a limited associativity design due to the following advantages: First, limited associativity caches are embarrassingly parallel and allow for concurrent operations on different sets without any form of synchronization (as illustrated in Figure 1). In contrast, it is far from trivial to introduce concurrency to most fully associative cache policies as often these cache management policies rank cached data items and then choose the "worst" item for eviction, c.f., [28, 39]. Thus, parallel actions may select the same cache victim or update the ranks of cached data items. For example, in LRU, we get contention on the head of the LRU list [20]. Aside from parallelism, cache implementations rely on elaborate data structures that often require an excessive number of pointers. Also, hash tables almost always have a constant fraction of unused space. In comparison, limited associativity caches offer a denser representation of data. When k is reasonably small, we can scan all the items in the set without requiring any auxiliary data structures.

The above limitations of full associativity have motivated the development of reduced accuracy management policies such as Clock [15, 25], sampled LRU [7] and sampled dynamic priority based policies [14, 12]. As elaborated in this work, we claim that limited associativity is a beneficial design alternative compared to Clock and sampling, especially for parallel systems.

1.1 Contributions

In this work, we explore and promote limited associativity designs for software caches. Specifically, our first contribution is the implementation of several leading cache replacement policies in a limited associativity manner. As we show, limited associativity leads to simplified data structures and algorithms. These can be trivially implemented in static memory with minimal overhead and operate in constant time. Further, these realizations trivially support wait-free concurrent manipulation and avoid hot-spots.

Our second contribution is an evaluation of the hit-ratio obtained by multiple cache management policies on real traces for k-way set associative caches and for fully associative caches. Our findings indicate that, even for relatively small values of k, the difference between the hit-ratio obtained by a k-way associative scheme and the corresponding fully associative scheme is marginal.

Since the runtime of each operation depends on the size of each set, low associativity is preferred for speed. Low associativity also implies more independent sets which reduces the contention on each set, and allows for better parallelism. However, this is a tradeoff as associativity also impacts the hit-ratio and if we set the associativity too low then we would suffer a non-negligible drop in hit-ratio. Our work demonstrates that a reasonable associativity value of 8 provides the best of both worlds as it yields a very similar hit-ratio to fully associative caches, while being considerably faster.

These results echo previous studies from hardware caching [22, 24], where limited associativity was found to be nearly as effective as full associativity. However, those studies were mostly conducted using hardware workloads, which tend to exhibit higher locality than software workloads.

We stress that limited associativity is a design principle rather than a cache management policy. Hence, we can adjust most existing policies to their respective limited associativity version with small changes. This assertion is especially straightforward for sampling based policies such as sampled LRU [7], Hyperbolic [14] and LHD [12] that randomly select a small number of cached items and choose the victim only from these items. Notice that limited associativity only requires computing a single hash function per cache miss compared to invoking the PRNG multiple times in sampled approaches. The sampled approach also requires accessing multiple random memory locations, which is not friendly for the hardware cache. In contrast, in limited associativity, we access a short continuous region of memory.

Further, contemporary cache management schemes, including ARC [33], LIRS [26], FRD [38] and W-TinyLFU [17, 16] maintain two or more cache regions, each of which handled in a fully associative manner. We argue that each cache region could be treated as a corresponding limited associativity region for these schemes.

Last, we evaluate the speedups obtained by our concurrent implementations compared to leading alternatives. These demonstrate the performance benefit of limited associativity designs for software caches in modern multi-core computers.

Paper Organization Section 2 surveys related work and provides the background necessary to position our work within the broader context of relevant designs. Section 3 suggests several ways to implement k-way set associative caches efficiently in software. Section 5 shows an evaluation of the hit ratio obtained by limited associativity caches for a wide selection of caching algorithms and of their throughput compared to the Guava and Caffeine Java caching libraries. We conclude with a discussion in Section 6.

2 Related Work

2.1 Cache Management Policies

A cache management scheme's design often involves certain (possibly implicit) assumptions on the characteristics of the "typical" workload. For example, the *Least Recently Used (LRU)* policy always admits new items to the cache and evicts the least recently used items [22]. LRU works well for *recency biased workloads*, where recently accessed items are most likely to be accessed again.

Alternatively, the *Least Frequently Used (LFU)* policy, also called *Perfect LFU*, assumes that the access distribution is fixed over time, meaning that frequently accessed data items are more likely to be reaccessed. Hence, Perfect LFU evicts the least frequently used item from the cache and admits a new one if it is more frequent than the cache victim. For synthetic workloads with static access distributions, Perfect LFU is the optimal cache management policy [27]. Real workloads are dynamic, so practical LFU policies often apply aging mechanisms to the items' frequency. Such mechanisms can be calculating frequency on sliding windows [27], using exponential decay [10, 9], or periodic aging [17].

Operation complexity is another essential factor in cache design. Both LRU and LFU need to maintain a priority queue or a heap, and can be implemented in constant time [29, 34, 13]. LRU and variants of LFU also maintain a hash table for fast access to items and their metadata. Another limitation of priority queue based LRU implementations is the high contention on the head of queue due to the need to update it on each cache access (both hits and misses).

The realization that a good cache management mechanism needs to combine recency and frequency led to the development of more sophisticated approaches [20]. LRU-K [37] maintains the last K occurrences of each item and orders all items based on the recency of their last K^{th} access. The drawbacks are

significant memory and computational overheads. Also, most benefit comes when $K = 2$, with quickly diminishing returns for larger values of K.

Low Inter-reference Recency Set (LIRS) [26] is a page replacement algorithm that attempts to directly predict the next occurrence of an item using a metric named *reuse distance*. To realize this, LIRS maintains two cache regions and a large number of ghost entries. FRD [38] can be viewed as a practical variant of LIRS, which overcomes many of the implementation related limitations of LIRS.

Window-TinyLFU (W-TinyLFU) [17] also balances between recency and frequnecy using two cache regions, a *window cache* and the *main cache*. Yet, it controls which items are admitted to the main cache using an optimized counting Bloom filter [19] based admission filter and thereby avoids the need to maintain ghost entries. The relative size of the two caches is dynamically adjusted at runtime [16].

As we mentioned before, what is common to ARC, CAR, LIRS, SLRU, 2Q, FRD and W-TinyLFU is that they all maintain two or more caches regions, each of which is maintained as a fully associative cache. We promote implementing each such cache region as a limited associativity cache.

Hyperbolic caching [14] logically maintains a dynamically evolving priority for each cached item, which is the number of times that item has been accessed divided by the duration of time since the item was inserted into the cache. To make this practical and efficient, priorities are only computed during the eviction and only for a small sample of items. Among the sampled items, the one whose priority is smallest becomes the victim. As we discuss later, this policy can be trivially implemented using an associative design.

2.2   Associativity in Caches

The only limited associativity software cache we are aware of is HashCache [11]. Yet, that work focused on very memory constraint Web caches in which the meta-data cannot fit in main memory and the goal is to minimize disk accesses to the meta-data. Further, it only considered the LRU replacement policy and only very particular workloads. In this paper, we perform a more systematic study of limited associativity's effectiveness to software cache design.

Current caching libraries use fully associative caches. Yet, in principle, any distributed cache (e.g., memcacheD) can be viewed as relying on limited associativity, since each node manages its caches independently to the others. When an item is added to a given cache, the victim must be selected from the same host. The main difference though is that in distributed caching, each host is typically responsible for a very large range of objects (equivalent to having a very large k value). In contrast, in k-way associativity, our goal, as explained before, is to have a large number of small sets (low k values). Further, the CPU's L1, L2 and L3 caches are almost always implemented through limited associativity, with the exception of TCAM based caches. In hardware, it is clear that fully associative caches require fewer circuits to implement and that they work faster.

2.3   On Hash Tables and Caches

As mentioned above, most fully associative software caches use fully associative hash tables to quickly find the cached items. In principle, hash tables incur non-negligible overheads. Specifically, open addressing hash tables such as the classic Linear Probing, or the modern Hopscotch hash table [23], are implemented on top of partially full arrays. In all open address hash tables, a constant fraction of these arrays must remain empty. Thus, relying on such tables results in underutilized space that could potentially store useful items. Closed address hash tables such as Java's concurrent hash map [4] are another alternative. Yet, they incur significant overheads for pointers, which results in a suboptimal data density.

In contrast, limited associativity caches can be implemented using the same arrays as open address hash tables, but we can use 100% of the array's cells. In principle, we do not even have to use pointers and achieve a very dense representation of data. Such a trade-off is potentially attractive since our caches can store more items within a given memory unit and having more items increases the cache hit ratio. On the other hand, there is less freedom in selecting the cache victim, which may reduce the hit ratio.

Let us emphasize that hashtables and caches are not the same. While a hashtable can be used to store cached objects, one would still need additional data-structures and algorithms to implement the cache management policy, and this is the main source of pain in software caches. On the other hand, caches are allowed to evict items while hashtables retain all the items that were not explicitly removed.

Figure 2: An example of a 48 items cache organized in a 6-way set associative design. The cache is broken into 8 independent sets such that each contains a up to 6 items.

2.4   Parallel Caches

Caches need to operate fast and thus parallelism is a vital consideration. Specifically, LRU is known to yield high contention on the head of the list as a simple implementation moves elements to the head of the list on each access. This contention motivated alternative designs such as Clock [15]. In Clock, each entry receives a single bit, indicating whether it was recently accessed. Items are always admitted by Clock and the cache victim is the first item with an unset Clock bit. The Clock hand clears set bits as it searches for the cache victim. Although the worst-case complexity of Clock is linear, it parallelizes well and creates little contention and is therefore used in practice as an operating system page replacement cache. Another interesting approach is taken by [18], which uses Bloom filter based approximations of LRU to enable better concurrency.

Fully associative cache management often relies on a hash table that needs to be parallelized. Such tables are a shared resource and are usually not embarrassingly parallel [35], [30]. On the other hand, limited associativity caches are embarrassingly parallel as actions on different sets are completely independent and require no synchronization. This property holds regardless of the (limited associativity) eviction policy.

3   Limited Associativity Architecture

A K-Way cache supports two operations: get/read and put/write, as customary in other caches such as Caffeine [31] and Guava [21]. A get/read operation retrieves an item's value from the cache or returns null if this item is not in the cache. The put/write operation inserts an item into the cache; if the item already exists in the cache, this operation overwrites its value. Both types of operations update the item's metadata that is used by the management policy.

Figure 2 illustrates such cache organization consisting of 8 independent sets that are held in an array. Each set is an array of 6 ways (nodes) that store items with metadata that may be required by various eviction policies.

Upon insertion of an item to the cache, we use a hash function to determine its set. If the set is full, we also need to select a victim from the same set to be evicted. K-Way supports efficient implementations of numerous eviction policies without relying on expensive auxiliary data structures. For example, to implement LRU or LFU, we only need a short counter for each cached item in the set. We select the victim by reading all K items in the set and evicting the one with the smallest counter. The difference between LRU and LFU is merely in the way the counter is updated. In LRU, we use it to record the (logical) timestamp of the last access, whereas in LFU, we use it to count how many times the entry was accessed in the past. To read an item, we scan the appropriate set. Once found, we update the corresponding item's metadata (updating the access time and frequency) atomically and returning the value.

Clearly, the K-Way time complexity of both get/read and put/write is O(K), which is only dependent on the number of ways (K). As K is a small constant number, we treat it as constant time.

Similarly, for Hyperbolic caching [14] we maintain two short counters for each item, one recording the time it was inserted (t_0) and the other (n) counting the number of times it was accessed (initialized to 1). For an eviction at time t_e, we scan the K items at the corresponding set and select the item whose $n/(t_e - t_0)$ value is minimal.

Algorithm 1 Internal KW-WFA classes and fields

```
1: class Node {
2:    K  key;
3:    V value;
4:    AtomicInteger counter;
5:    int index; }
6: class set {
7:    AtomicReferenceArray<Node<K, V>> setArray;
8:    AtomicLong time; } // time is only present in LRU
```

Algorithm 2 KW-WFA get(read) operation

```
1: function GET(K key)
2:    int set =(int) hash(key) & (numberOfSets-1);
3:    for int i = 0; i < ways; ++i do
4:        Node<K, V> n = cache[set].setArray.get(i);
5:        if  n != null && n.key.equals( key) then
6:            update(n.counter);
7:            return n.value;
8:    return null;
```

We implemented K-Way in Java. It supports five eviction policies: LRU, LFU, FIFO, Random and Hyperbolic. The eviction policy is chosen at the constructor. We have realized several concurrency control variants as detailed below.

K **Way Cache Wait Free Array - KW-WFA** Each set is an array of Node references. Algorithm 1 depicts the internal classes and variables. Pseudo-code for the get operation is given in Algorithm 2 and for the put operation in Algorithm 3. To replace a victim node with a newly arriving one, we perform a CAS (compare and swap) operation on the address of this node.

To exchange an entire node atomically, we maintain each set of K-Way as a reference array. Alas, this has the following shortcoming. To search an item and select a victim, we must scan the entire corresponding set reading the metadata of each node. With a reference array implementation, this means reading multiple different memory locations, one for each item in the set, which is expensive.

To improve this and enable scans to access continuous memory regions, we offer a second implementation in which we separated the counters and fingerprints from the nodes.

K Way Cache Wait Free Separate Counters - KW-WFSC We store an array of counters and an array of fingerprints with the reference array of the nodes for each set. Algorithm 4 depicts the internal classes and variables. To insert an item, we first scan the fingerprints array. If a certain entry's fingerprint matches the key's fingerprint, we check the key inside the corresponding node. If they are the same, we update this node with the new value and return. If the item is new, we scan the counter array to find the lowest/highest counter, depending on the eviction policy. We then replace the victim without accessing the node at all. Pseudo-code for the get operation is listed in Algorithm 5 and for the put operation can be found in Algorithm 6.

K **Way Cache Lock Set - KW-LS** In our third implementation, we utilize one lock per set. Algorithm 7 depicts the internal classes and variables. To read an item from the set, we lock the set and scan it. If the item is in the set, we try to change the lock to a write lock to update the counter, return the value and finish. We treat writes in a similar manner. Pseudo-code for the get operation is given in Algorithm 8 and for the put operation in Algorithm 9.

4   Formal Analysis

Assume that a fully associative cache policy would like to store C items in the cache. We can guarantee that any specific C items can be kept in a larger limited associativity cache. Specifically, if we denote the number of sets n, then if $C \geq n \log(n)$ we get that the maximum load (among the C items) is:
$\frac{C}{n} + \Theta\left(\sqrt{\frac{C \log(n)}{n}}\right)$.

Algorithm 3 KW-WFA put(write) operation

```
1: function PUT(K key , V value)
2:     int set =(int) hash(key) & (numberOfSets-1));
3:     for int  i = 0; i < ways; ++i do
4:         Node<K, V> n = cache[set].setArray.get(i);
5:         if n != null  &&  n.key.equals( key) then
6:             cache[set].setArray.compareAndSet(i,n,update);
7:             return ;
8:     Node<K, V> victim =policy.select(cache[set].setArray, key)
9:     if  victim != null  then
10:        Node<K,V> updateNode = new Node<K,V>(key,value,victim.index,cache[set].readTime());
11:        cache[set].setArray.compareAndSet(i,victim,update);
12:    else
13:        for int i = 0; i < ways; ++i do
14:            if Node<K, V>) cache[set].setArray.get(i)== null then
15:                Node<K, V> updateNode = new  Node<K, V>(key,value,victim.index,cache[set].readTime());
16:                cache[set].setArray.compareAndSet(i ,null,updateNode);
17:                return;
```

Algorithm 4 Internal KW-WFSC classes and fields

```
1: class Node {
2: K key;
3: V value;
4: AtomicInteger counter;
5: int index; }
6: class set {
7: AtomicReferenceArray<Node<K, V>> setArray;
8: AtomicIntegerArray counters;
9: AtomicLongArray fingerPrint; }
```

The next theorem guarantees that a K-way set associative cache can mimic a (smaller) fully associative cache with high probability. Thus, such caches are capable of comparable performance as their fully associative counterparts. Our theorem uses the following Chernoff bound (Theorem 4.4, page 66) from [36]: Let $X_1, X_2, ... X_n$ be independent Poission trails, with $E(X_i) = p_i$, let $X = \Sigma_{i=1}^n X_i$ and let $\mu = E(X)$ then: $\Pr[X \geq (1+\delta)E(X)] \leq e^{\frac{-E(X)\delta^2}{3}}$.

Theorem 4.1. *Consider a K-Way set associative cache of size $C' \geq 2C$, then the probability for the cache to be able to store the desired C' items is at most:* $\frac{C'}{k}e^{\frac{k}{6}}$.

Proof. When storing C items in a C' sized limited associativity cache partitioned into $\frac{C'}{k}$ sets, each set receives $\frac{Ck}{C'} = \frac{k}{(1+\delta)}$ of those C items. We look at an arbitrary set and denote X_i the random variable of the ith item. That is, $X_i = 1$ if the $i'th$ item is placed on the set in question and zero otherwise. We denote by X, the total number of items (among the C) that are placed in the set. By definition, we get that: $E(X) = \frac{k}{(1+\delta)}$.

From the Chernoff bound we get: $\Pr[X \geq (1+\delta)E(X)] \leq e^{\frac{k\delta^2}{(1+\delta)3}}$. That is, the probability for each set to not be able to store one of the items allocated to it is: $e^{\frac{k\delta^2}{(1+\delta)3}}$. We apply the union bound on all sets and obtain that the probability that none of the sets receives more items than it can store is at most: $\frac{C'}{k}e^{\frac{k\delta^2}{(1+\delta)3}}$. Substituting $\delta = 1$ concludes the proof. ☐

For example, Theorem 4.1 shows that a 64-way cache of size 200k items can store any desired 100k items with a probability of over 99%. Alternatively, we can store in a 2M sized 128 way set associative cache any 1M items with a probability of over 99.999%.

In practice, even when limited associativity caches miss out on some desired items, they store enough other items to yield comparable hit ratios as same size fully associative caches. This is due to the following two reasons: First, the above bound is not tight. Second, it is quite reasonable that slight content divergence between caches would still result in similar hit ratios. That is, the exact cache content is just a mean to obtain good hit ratio, not a goal by itself.

Algorithm 5 KW-WFSC get operation

```
1: function GET(K key)
2:     int set =(int) hash(key) & (numberOfSets-1);
3:     for int i = 0; i < ways; ++i do
4:         Long fp = cache[set].fingerPrint.get(i);
5:         if fp.equals((Long)key) then
6:             Node<K, V> n = cache[set].setArray.get(i);
7:             if n != null && n.key.equals( key) then
8:                 update(n.counter);
9:                 return n.value;
10:    return null;
```

Algorithm 6 KW-WFSC put operation

```
1: function PUT(K key, V value)
2:     int set =(int) hash(key) & (numberOfSets-1));
3:     for int i = 0; i < ways; ++i do
4:         Long fp = cache[set].fingerPrint.get(i);
5:         if fp.equals((Long)key) then
6:             Node<K, V> n = cache[set].setArray.get(i);
7:             if n != null && n.key.equals( key) then
8:                 Node<K, V> update = new Node<K, V>(key, value, i);
9:                 cache[set].setArray.compareAndSet(i, n, update);
10:                return;
11:    Node<K,V> victim =policy.select(cache[set].counters, key);
12:    Node<K,V> update = new Node¡K,V¿(key,value,victim.index,cache[set].readTime());
13:    if cache[set].setArray.compareAndSet(victim.index ,victim,update); then
14:        cache[set].fingerPrint.set(victim,index,(Long)key);
15:        cache[set].counters.set(victim,index, 0);
```
---

5 Evaluation

5.1 Workloads and Setup

We utilized the following real world workloads:

wiki1190322952 and wiki1191277217: two traces from Wikipedia containing 10% of the traffic to Wikipedia during three months starting in September 2007 [43].

sprite: From the Sprite network file system, which contains requests to a file server from client workstations for a two-day period by [26].

multi1: This trace is obtained by executing two workloads, cs, an interactive C source program examination tool trace, and cpp, a GNU C compiler pre-processor trace by [26].

multi2: This is obtained by executing three workloads, cs, cpp and postgres, a trace of join queries among four relations in a relational database system from the UC Berkeley, provided by [26].

multi3: This is obtained by executing four workloads, cs, cpp, glimpse, a text information retrieval utility, and postgres provided by [26].

OLTP: A file system trace of an OLTP server [33].

DS1: A database trace provided by [33].SMALL

S3: Search engine trace provided by [33].

S1 and S3: Search engine traces provided by [33].

P8, P12 and P14: Windows server disc accesses [33].

F1 and F2: Traces of transaction processing taken from large financial institution. The trace is provided by the UMass trace repository [44].

W2 and W3: Search engine traces provided by [44].

Algorithm 7 Internal KW-LS classes and fields

```
1: class Node {
2:    K key;
3:    V value;
4:    Integer counter; }
5:    int index; }
6: class set {
7:    Node < K, V > []setArray;
8:    StampedLock lock; }
```

Algorithm 8 KW-LS get operation

```
1: function GET(K key)
2:     int set =(int) hash(key) & (numberOfSets-1));
3:     long stamp =cache[set].lock.readLock();
4:     for  int i = 0; i < ways; ++i do
5:        Node<K, V> n = cache[set].setArray.get(i);
6:        if n != null && n.key.equals( key) then
7:           long stampConvert=cache[set].lock.tryConvertToWriteLock(stamp);
8:           if  stampConvert == 0  then
9:              cache[set].lock.unlockRead(stamp);
10:              return n.value;
11:           update(n.counter);
12:           cache[set].lock.unlockWrite(stampConvert);
13:           return n.value;
14:     cache[set].lock.unlockRead(stamp);
15:     return null;
```

In all figures below, the "k ways" and "sampled" lines are K-Way with set/sample sizes of $4, 8, 16, 32, 64$ & 128. The "fully associative" line stands for a linked-list based fully associative implementation. The "product" figures for each trace present Guava cache by Google [21], Caffeine [31] and segmented Caffeine [32], which is Caffeine divided into 64 segments to enable Caffeine to benefit from parallelism at the possible expense of reduced hit-ratios (plain Caffeine serves updates by a single thread).

In all throughput evaluations, we compare K-Way with $k = 8$ to sampled implementations of corresponding methods with sample size $= 8$, Guava cache, Caffeine and segmented Caffeine. All evaluated cache implementations are written in Java. Our K-Way cache uses xxHash [42] as the hash function that distributes items to sets.

5.1.1   Platform

We ran on the following platforms:

- AMD PowerEdge R7425 server with two AMD EPYC 7551 Processors, each with 32 2.00GHz/2.55GHz cores that multiplex 2 hardware threads, so in total, this system supports 128 hardware threads. The hardware caches include 32K L1 cache, 512K L2 cache and 8192K L3 cache. It has 128GB DRAM organized in 8 NUMA nodes, 4 per processor.

- Intel Xeon E5-2667 v4 Processor including 8 3.20GHz cores with 2 hardware threads, so this system supports 16 hardware threads. It has 128GB DRAM. The hardware caches include 32K L1 cache, 256K L2 cache and 25600K L3 cache.

5.1.2   Methodology

During throughput evaluation, we employed the following typical cache behavior: For each element in the trace, we first performed a read operation. If the element was not in the cache, we initiated a write operation of that element into the cache.

We started with a warm-up for filling the cache with elements not in the trace. We did the warm-up first in the main thread that inserted elements up to the cache size and then in each thread by inserting $size/\#threads$ non cached elements that are not in the trace. All threads started the test after the warm-up simultaneously by waiting on a barrier.

We ran each test for a fixed period, counting the number of operations. The time was calculated in comparison to the size of the trace and is between 1 and 4 seconds. Each trace was run with a different cache size taking into account the size of the trace and the hit rate. Each point in the graphs is the mean taken over 11 runs.

Algorithm 9 KW-LS put operation

```
1:  function PUT( K key , V value)
2:      int set =(int) hash(key) & (numberOfSets-1));
3:      long stamp =cache[set].lock.readLock();
4:      for  int i = 0; i < ways; ++i do
5:          Node<K, V> n = cache[set].setArray[i];
6:          if  n != null && n.key.equals( key) then
7:              long stampConvert=cache[set].lock.tryConvertToWriteLock(stamp);
8:              if  stampConvert == 0 then
9:                  cache[set].lock.unlockRead(stamp);
10:                  return ;
11:             n.value=value;
12:             update(n.counter)
13:             cache[set].lock.unlockWrite(stampConvert);
14:             return;
15:     Node<K, V> victim =policy.select(cache[set].setArray, key)
16:     int  victimIndex = 0
17:     if victim != null  then
18:         victimIndex = victim,index
19:     else
20:         for  int i = 0; i < ways; ++i  do
21:             if Node<K, V> cache[set].setArray.get(i)== null then
22:                 victimIndex =i
23:     Node<K, V> updateNode = new Node<K, V>(key,value,victim.index,cache[set].readTime());
24:     cache[set].setArray[victimIndex].key=key;
25:     cache[set].setArray[victimIndex]value.=value;
26:     cache[set].setArray[victimIndex].counter=0;
27:     cache[set].lock.unlockWrite(stampConvert);
28:     return;
```


Figure 3: Legend for all graphs.

5.2 Hit Ratio Evaluation

Figure 3 shows the legend for all the hit ratio and throughput graphs shown in this work. Figures 4 and 6 to 13 depict the results for the traces described above with different eviction and admission policies. For brevity, not all traces and combinations are shown. The ones removed essentially exhibit similar results.

The first graphs for all traces is LRU due to its popularity. The second graph is LFU eviction with TinyLFU admission, first proposed in [17]. LFU is also a popular policy, but LFU alone is not a very good management policy as it lacks aging mechanisms and lacks data about non-cached objects, which are added by the TinyLFU admission mechanism. The third graph includes the real caching products: Guava [21], Caffeine [31] and the proof-of-concept segmented Caffeine (with number of independent segments that match the number of threads tested) [32]. For brevity, the fourth graph presents different (additional) policies for different traces. We summarise the results below:

LRU eviction policy Recall that LRU evicts the least recently used item (and admits all items). In the limited associativity model, we first assign a new item to one of the small-sized sets based on hashing its key and then evict the least recently used item of that set. As can be seen from the graphs, for most traces, the level of associativity has minimal impact on the obtained hit ratio.

LFU eviction with LFU admission The results for this policy (in subfigures b) show that associativity has a minor impact on the hit ratio. Also, observe that the performance of the sampled approach is similar to that of limited associativity. In some cases one is slightly better than the other and in others it is the opposite.

Figure 4: wiki1190322952.

Products In these graphs (subfigures c), we compared Guava cache [21], Caffeine [31] and segmented Caffeine [32]. In the latter, we used a hash function to map each element to one of the corresponding Caffeine managed segments. Each such Caffeine instance was constructed with MAX SIZE/#$threads$, to hold MAX SIZE elements in total. As can be observed, the hit-ratio of segmented Caffeine is nearly identical to that of Caffeine. Also, notice that Caffeine attains higher hit-ratio than Guava as it uses the climber policy from [16] compared to some version of LRU in Guava.

Hyperbolic Caching We also show that limited associativity has little effect on Hyperbolic caching (figs. 6 and 12) and on Hyperbolic caching with the TinyLFU admission policy (fig. 8). Notice that for Hyperbolic caching, there are slight differences between sampling and limited associativity. When combining Hyperbolic with TinyLFU admission, these differences become barely noticeable.

Conclusions: This motivational study concludes that limited associativity only has a minor impact on the obtained hit ratio.

5.3   Throughput Evaluation on Real Traces

The throughput measurements, in Millions of Get/Put operations per second, with different number of threads are shown in Figures 14 to 18 for the AMD PowerEdge R7425 server and figs. 19 to 26 for the Intel Xeon E5-2667 server.

As can be observed, the K-Way variations KW-WFA, KW-WFSC and KW-LS gain the highest throughput. Caffeine and sampled exhibited the worst worst-case performance, and it does not improve with additional threads for all traces except Sprite (fig. 24), which will be addressed shortly.

For Caffeine, this is explained as put operations in Caffeine are performed in the background by a single thread. Guava is considerably faster than Caffeine in traces with significant number of misses because it performs put operations in the foreground in parallel. As for the sampled approach, on every

(a) LRU (b) LFU +TinyLFU (c) Product (d) Hyperbolic

Figure 5: P8.

miss it needs to calculate K random numbers and access these K random locations to create the sample. Even when K is as small as 8, this already incurs a significant overhead.

Our limited associativity design therefore serves misses much faster than sampled (and Caffeine). However, sampled may be quicker in serving a hit since it only accesses the (single) accessed object's meta-data, while our limited associativity design always accesses the entire set. Further, the larger the cache is, the higher is the number of sets (since K is independent of the cache size). Hence, the hash function of the limited associativity approach is likely to yield better load balancing for larger caches.

For Sprite (fig. 24), the hit ratio is high and the caches are small. Hence, our limited associativity under-performs compared to sampled. Yet, it still delivers tens of millions of operations per second.

5.4 Synthetic Throughput Evaluation

100% miss ratio: The worst-case performance for any cache is the case of 100% misses. In this evaluation, we perform a single get and a single put operation for each item and each item is requested only once. This extreme case helps us characterize the operational framework of various cache libraries.

Figure 27 shows the attained throughput (measured in Millions of Get/Put operations per second). As can be observed, Caffeine has the worst worst-case performance and it does not improve with additional threads. This is explained as put operations in Caffeine are performed in the background by a single thread. Guava is considerably faster than Caffeine, because it performs put operations in the foreground, and next are our own three implementations of limited associativity caches. Here, the KW-WFSC is the fastest of our algorithms because it is optimized towards fast replacement of the cache victim.

100% hit ratio: Next, we repeat the last experiment but only perform Get operations to mimic having 100% hit ratio. Our results are in Figure 28. Notice that here, Caffeine is considerably faster than all the alternatives since its read operations are simple hash table read operations. Next is Guava, which is very fast, and our algorithms are last. Interestingly, our algorithms attain very similar performance to the 100% miss case, as we always scan the set - a full scan on a miss and half the set on average for a hit.

Figure 6: P12.

This implies that their performance is less sensitive to the workload characteristics. The performance of Caffeine and Guava varies greatly according to the cache effectiveness. We conclude that the break-even point between our approach vs. Caffeine and Guava depends on the hit ratio of the trace. So our next evaluation aims to determine at what hit ratios limited associativity caches are expected to outperform these established libraries.

95% hit ratio: We continue and simulate 95% hit ratio by performing one put operations for every 20 read operations. The results in Figure 29 show that in 95% hit ratio Guava is the fastest algorithm, while Caffeine is fastest for a small number of threads. Here, again Caffeine does not scale with the number of threads because the bottleneck still becomes the write buffer.

90% hit ratio: Next, we increase the number of writes to 1 in 10 to simulate 90% hit ratio. The results in Figure 30 show that KW-WFSC is already faster than Guava and Caffeine when there are enough threads. That is, we conclude that our approach attains higher throughput than Caffeine and Guava as long as the hit ratio is less than 90%, which implies our attractiveness for a very large variety of workloads and cache sizes.

Conclusions: In our limited associativity design, get and put have similar performance. In contrast, in sampled, Caffeine and Guava gets are fast and puts are slow. Our performance is expected to be better than sampled, Caffeine and Guava when the hit ratio is below 90%, which captures most real traces and cache sizes. Our approach scales better with the number of threads in almost all scenarios.

6 Discussion

Our work argues that software cache associativity is an important design choice that should be given consideration. We note that limited associativity caches are easier and simpler to implement than fully

Figure 7: S1.

associative caches. They are embarrassingly parallel and have lower memory overheads compared to fully associative caches. On the other hand, limited associativity may negatively impact the hit ratio, which is an important quality measure for every cache.

Sampled techniques, such as Sampled LRU of Redis [7] and the sampling performed by Hyperbolic caching [14], reduce accuracy in favor of speed. This is because one can always improve the hit ratio by increasing the cache size, but there are no simple techniques to improve the operation speed of the cache. On the other hand, caches' utility is due to performance gaps between different ways to fetch data. For example, main memory is faster than secondary storage. Therefore, the potential benefits from caching are based on the gap between the operation time of the cache and the operation time of the alternative method. Thus, improving the operation time of caches makes every cache hit more beneficial to the application (as long as we do not significantly reduce the hit ratio).

Our work evaluated limited associativity caches on diverse (real) application workloads over multiple popular cache management policies. We showed that limited associativity caches attain a similar (and often nearly identical) hit ratio compared to fully associative caches for many workloads and policies. Our work also suggested three different implementations of limited associativity caches. We showed an improvement of up to x5 compared to the Caffeine and Guava libraries that together dominate the Java ecosystem, especially in multi-threaded settings. Given these findings, we believe that such caching libraries would benefit from adopting limited associativity cache designs.

Among our different implementations, when the trace is uniformly separated without heavy hitters, so the insertions are usually to different buckets, it is better to use the KW-LS. If there are many reads and not many writes, it is better to use the KW-WFSC as the reads are continuous in memory and hence faster than KW-WFA where the array is an array of pointers so the performance of the reads is compromised. Otherwise, it is best to use the KW-WFA as it uses only one atomic operation in contrast to KW-WFC with three atomic operations, which slow down the cache updates.

Interestingly, for hardware caches, the seminal work of Qureshi, Thompson and Patt [40] has found that to fully optimize the cache hit ratio, one should vary the associativity of a cache on a per-set basis in response to the demands of the program. Supporting such dynamic associativity level in software is

14

Figure 8: S3.

straight-forward and could help improve the hit-ratio gaps between 8 ways and fully associative in traces like WS2. Exploring this direction is left for future work.

References

[1] Apache Accumulo. 2020.

[2] Apache Cassandra. 2020.

[3] Apache HBase. 2020.

[4] Java ConcurrentMap. https://en.wikipedia.org/wiki/Java_ConcurrentMap, 2020.

[5] The DGraph Website. 2020.

[6] The Neo4j Website. 2020.

[7] Using Redis as an LRU Cache. 2020.

[8] S. Akhtar, A. Beck, and I. Rimac. HiFi: A Hierarchical Filtering Algorithm for Caching of Online Video. In *ACM MM*, MM, pages 421–430, 2015.

[9] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *In Proc. of the 2nd Workshop on Internet Server Performance*, 1999.

Figure 9: OLTP.

[10] M. Arlitt, R. Friedrich, and T. Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. *Perform. Eval.*, 39(1-4):149–164, Feb. 2000.

[11] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache Storage for the Next Billion. In *NSDI*, pages 123–136, 2009.

[12] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *NSDI*, pages 389–403, Apr. 2018.

[13] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy Hitters in Streams and Sliding Windows. In *IEEE INFOCOM*, 2016.

[14] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *ATC*, pages 499–511, 2017.

[15] F. J. Corbato. A Paging Experiment with the Multics System. Technical Report Project MAC Report MAC-M-384, MIT, May 1968.

[16] G. Einziger, O. Eytan, R. Friedman, and B. Manes. Adaptive Software Cache Management. In *Middleware*, pages 94–106, 2018.

[17] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM ToS*, 2017.

[18] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, pages 371–384, 2013.

[19] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[20] R. Friedman and B. Manes. Tutorial: Designing Modern Software Caches. FAST, 2020.

Figure 10: LIRS - multi2.

[21] Google. Guava: Google Core Libraries for Java. *https://github.com/google/guava*, 2016.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

[23] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. DISC, pages 350–364, 2008.

[24] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec 1989.

[25] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *USENIX Annual Technical Conference*, ATEC, pages 35–35, 2005.

[26] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *ACM SIGMETRICS*, pages 31–42, 2002.

[27] G. Karakostas and D. N. Serpanos. Exploitation of Different Types of Locality for Web Caches. In *ISCC*, 2002.

[28] R. Karedla, J. S. Love, and B. G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, 1994.

[29] A. Ketan Shah and M. D. Matani. An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme. Technical report, 2010. "http://dhruvbird.com/lfu.pdf".

[30] T. Maier, P. Sanders, and R. Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, 2019.

[31] B. Manes. Caffeine: A High Performance Caching Library for Java 8. *https://github.com/ben-manes/caffeine*, 2017.

[32] B. Manes. Segmented Caffeine – Private Communications. 2020.

Figure 11: multi3.

[33] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, pages 115–130, 2003.

[34] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*, 2005.

[35] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[36] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[37] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *ACM SIGMOD Rec.*, 22(2):297–306, June 1993.

[38] S. Park and C. Park. FRD: A Filtering based Buffer Cache Algorithm that Considers both Frequency and Reuse Distance. In *MSST*, 2017.

[39] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, Dec. 2003.

[40] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-Based Associativity via Global Replacement. In *ISCA*, pages 544–555, June 2005.

[41] G. Tewari and K. Hazelwood. Adaptive web proxy caching algorithms. Technical Report TR-13-04, "Harvard University", 2004.

[42] V. Tolstopyatov. xxHash. *https://github.com/OpenHFT/Zero-Allocation-Hashing*.

[43] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks*, 53(11):1830–1845, 2009.

Figure 12: sprite.

[44] G. Weikum. UMassTrace Repository. *http://traces. cs. umass. edu/index. php/Storage*, 2011.

Figure 13: WebSearch3.

Figure 14: F1 trace with cache size of 2^{11} elements and duration of run of 1 second, run on AMD PowerEdge R7425.

Figure 15: S3 trace with cache size of 2^{19} elements and duration of run of 4 second, run on AMD PowerEdge R7425.

Figure 16: S1 trace with cache size of 2^{19} elements and duration of run of 4 second, run on AMD PowerEdge R7425.

Figure 17: wiki1190322952 trace with cache size of 2^{11} elements and duration of run of 1 second, run on AMD PowerEdge R7425.

Figure 18: OLTP trace with cache size of 2^{11} elements and duration of run of 1 second, run on AMD PowerEdge R7425.

Figure 19: F2 trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 20: W3 trace with cache size of 2^{19} elements and duration of run of 4 second, run on Intel Xeon E5-2667.

Figure 21: multi1 trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 22: multi2 trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 23: multi3 trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 24: sprite trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 25: P12 trace with cache size of 2^{17} elements and duration of run of 2 second, run on Intel Xeon E5-2667.

Figure 26: wiki1191277217 trace with cache size of 2^{11} elements and duration of run of 1 second, run on Intel Xeon E5-2667.

Figure 27: Synthetic trace of read and write, 100% miss ratio, with cache size of 2^{21} elements, and duration of run of 5 second, run on AMD PowerEdge R7425.

Figure 28: Synthetic trace to mimic 100% hit ratio with cache size of 2^{21} elements, and duration of run of 5 second, run on AMD PowerEdge R7425.

Figure 29: Synthetic trace to mimic 95% hit ratio with cache size of 2^{21} elements, and duration of run of 5 second, run on AMD PowerEdge R7425

Figure 30: Synthetic trace to mimic 90% hit ratio with cache size of 2^{21} elements, and duration of run of 5 second, run on AMD PowerEdge R7425