Concurrent Hash Maps Under Pressure: Revisiting the Separate Chaining Mechanism with Linked Lists and Dynamic Arrays

Ana Castro, Miguel Areias, and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{ana-castro,miguel-areias,ricroc}@dcc.fc.up.pt

Abstract. Hash maps are a widely used and efficient data structure for storing and accessing data. One of the main challenges in hash map implementation is the management of collisions. Arguably, *separate chaining* is among the most well-known strategies for collision resolution. In this paper, we present a comprehensive study comparing two common approaches to implementing separate chaining - *linked lists* and *dynamic arrays* - in a multithreaded environment using a lock-based concurrent hash map design. Our study includes a performance analysis covering parameters such as cache behavior, energy consumption, contention under concurrent access, and resizing overhead. Experimental results show that dynamic arrays maintain more predictable memory access and lower energy consumption in multithreaded environments.

1 Introduction

Hash maps [15] are widely valued for their nearly constant average-case time complexity of $O(1)$ for insertion, deletion, and lookup operations. As a result, they play a crucial role in a broad range of applications, including symbol tables [2], dynamic programming [5], and database indexing mechanisms [13]. A key aspect of hash map design is the management of collisions. *Separate chaining* [16, 12] is a widely adopted strategy for collision resolution, typically implemented using either *linked lists* or *dynamic arrays*.

Concurrent hash maps aim to retain the advantages of their non-concurrent counterparts while ensuring correctness and high performance in multi-threaded environments [8]. In particular, a good collision management strategy is vital under concurrent hash maps designs [14]. Performance evaluation of concurrent hash maps typically focuses on throughput, latency, and scalability. Common metrics include operations per second, speedup relative to sequential baselines, and contention overhead. In addition, energy efficiency has become an increasingly important metric, particularly on architectures with deep memory hierarchies and non-uniform memory access patterns [10]. However, to the best of our knowledge, there is no comprehensive study that performs an in-depth comparison between separate chaining mechanisms, linked lists-based and dynamic

arrays-based, within a context of a lock-based concurrent hash map design. Despite their widespread use, the trade-offs between these two approaches - particularly in terms of cache behavior, contention under concurrent access, and resizing overhead - remain largely unexplored.

The remainder of the paper is organized as follows. We begin by introducing the necessary background and providing an overview of the two separate chaining approaches. Next, we provide a detailed description of the key algorithms we implemented from scratch to facilitate the reproduction of our work by others. We then present a comprehensive experimental study designed to evaluate the performance of both approaches. To this end, we employ measurement tools such as Intel's Running Average Power Limit (RAPL) [11] and Linux performance counters [1] to quantify relevant metrics, with particular emphasis on analyzing the energy and cache performance trade-offs. Finally, we conclude by summarizing our contributions and outlining potential directions for future work.

2 Background

Hash maps are a common and efficient data structure for storing and accessing data organized as key-value pairs. The mapping between a key K and a value V is provided by a *hash function*, which deterministically maps K to a specific index (or *bucket*) within an array-based structure. This bucket indicates the location where the corresponding value V is stored.

One of the main challenges in hash map implementation is the management of collisions, which occur when different keys are mapped to the same bucket. Common strategies for collision resolution include *separate chaining*, where each bucket references a secondary data structure to store multiple entries, and *open addressing*, in which alternative locations within the array are systematically probed to resolve conflicts [16, 12]. Chaining remains effective under moderate to high *load factors* (the ratio of stored elements to available buckets), although it introduces additional memory overhead and may degrade performance when chains grow long. Open addressing uses less memory but is more sensitive to clustering and requires careful tuning of the load factor to maintain performance.

When implementing chaining, *linked lists* and *dynamic arrays* are two common approaches. Linked lists offer stable performance (linear complexity under key collisions), but they tend to scale poorly on modern hardware architectures - characterized by load/store execution models and multi-level caches - due to poor spatial locality. Linked list nodes are not stored contiguously in memory, and pointer dereferencing incurs frequent cache misses. Dynamic arrays mitigate this problem to some extent by offering better spatial locality and cache performance. However, they introduce additional complexity in resizing and shifting operations, particularly as the number of elements grows or when insertions are highly interleaved with lookup and delete operations [7, 18].

We refer to the dynamic growth characteristic of separate chaining approaches as *horizontal expansion*. In contrast, when the load factor exceeds a predefined threshold, a *vertical expansion* occurs, involving the allocation of a new, larger

array of buckets (often sized to a prime number or a power of two), followed by the rehashing of all existing keys. Although vertical expansion incurs significant cost during the reallocation and rehashing process, this overhead is amortized over time, enabling the hash map to maintain constant average-case time complexity in the long run.

In concurrent environments, a major challenge is maintaining both correctness and performance during vertical expansion. Common strategies include double-buffering, where a new hash map is constructed in parallel and buckets are migrated one by one; and incremental expansion, where the rehashing is performed gradually as threads access individual buckets.

3 Separate Chaining Designs by Example

This section outlines the design choices behind our concurrent hash map implementation of the two alternative separate chaining approaches for collision resolution. Both approaches share the same underlying structure of a hash array of buckets and differ only in how the chaining mechanism is implemented.

3.1 Linked Lists

We begin by presenting the key aspects of the linked list approach. Figure 1 illustrates a simple example of how concurrent insertion is handled in this context. The figure depicts the standard hash map configuration, which is formed by a header structure that stores common information such as the number of bucket entries or the number of key-value pairs currently stored in the hash. It is also formed by a bucket array of entries, where each bucket contains a lock field L and a pointer reference. A bucket entry begins with a *Null* reference, and during execution it can store either a reference to a second hash level, if the current hash has been (vertically) expanded, or a reference to a chain of nodes representing hash collisions for that entry.

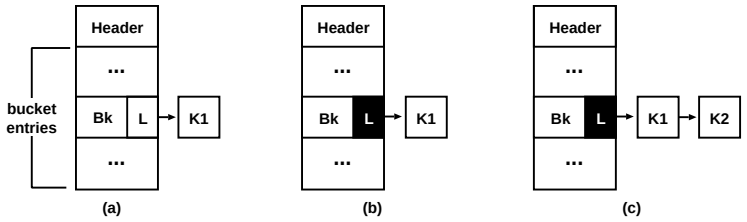

Fig. 1. Concurrent insertion with linked lists

Figure 1(a) shows that B_k represents a particular bucket entry that already contains a node with key K_1. For simplicity, only the keys are shown in the figures. Figure 1(b) shows the hash configuration before inserting a new node in B_k, which requires acquiring the lock for the bucket (represented by the black background). Figure 1(c) shows the hash configuration after inserting node K_2 in

B_k and before releasing the lock. New nodes are inserted at the end of the chain. Each node contains a reference to the next node in the chain and the last node contains a *Null* reference. When the number of nodes in a chain reaches a given threshold, the hash map is checked for vertical expansion. If the total number of nodes stored in and registered with the hash header exceeds a predefined load factor, the hash map expands to a second hash level. Figure 2 shows how nodes are concurrently expanded to a second hash level.

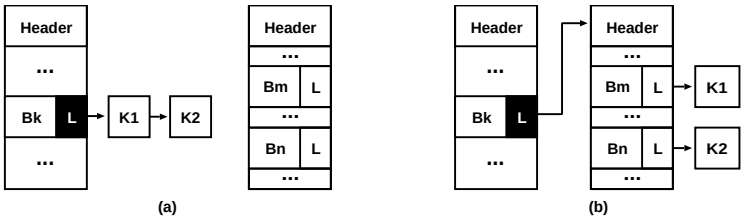

Fig. 2. Concurrent vertical expansion with linked lists

The thread responsible for performing vertical expansion begins by allocating a new hash level with twice the number of bucket entries. It then iterates over all buckets in the original array to rehash and redistribute the existing keys into the new level. For each bucket, the thread acquires the corresponding lock and transfers the chain nodes, one by one, to the appropriate buckets in the new hash. Figure 2(a) illustrates the configuration after acquiring the lock on bucket B_k, but before moving nodes K_1 and K_2 to the new hash level. Figure 2(b) shows node K_1 being moved to bucket B_m and node K_2 to bucket B_n in the new level. Once all nodes have been moved, bucket B_k is updated to reference the new hash level, indicating that future operations on B_k should be performed at the new level from this point onwards.

Once all buckets have been processed, the global entry point of the hash map is also updated to reference the new hash level, ensuring that all subsequent operations are performed on the new level.

Regarding the deletion operation, it also begins by acquiring the lock for the corresponding bucket. The chain is then traversed in search of the target key to be deleted. If a node N containing the key is found, N is deleted from the chain, the chain is updated accordingly, and N is subsequently freed.

3.2 Dynamic Arrays

For the dynamic array approach, we also begin with a simple example that illustrates how concurrent insertion works, as shown in Fig. 3. This figure depicts the same hash map structure as before. As with linked lists, each bucket entry initially contains a *Null* reference. During execution, this reference may be updated to point either to a second hash level or to a dynamic array representing hash collisions for that entry. In addition, bucket entries in this approach include two numeric fields: one indicating the size of the dynamic array (zero if no array is allocated), and another representing the number of elements stored in it.

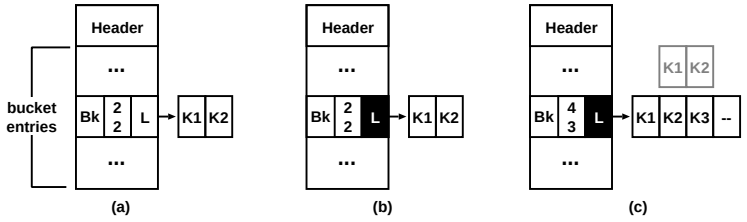

Fig. 3. Concurrent insertion with dynamic arrays

Figure 3(a) illustrates a bucket entry B_k that already contains a fully allocated dynamic array of size 2, storing keys K_1 and K_2. Figure 3(b) shows the hash map configuration just before inserting a new key K_3, which requires acquiring the lock for that bucket. Finally, Fig. 3(c) depicts the configuration after inserting K_3 into B_k and before releasing the lock. Since the original array was full, a new array with double the capacity (size 4 in this example) had to be allocated (horizontal expansion). The existing elements were copied into the new array, key K_3 was inserted, and the old array was then released. As before, when the number of elements in a dynamic array reaches a predefined threshold, the hash map checks whether vertical expansion is necessary. Figure 4 illustrates how concurrent vertical expansion is handled with dynamic arrays.

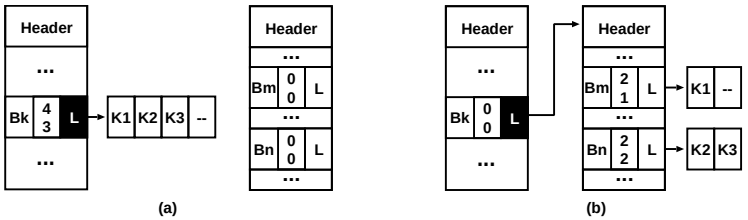

Fig. 4. Concurrent vertical expansion with dynamic arrays

Figure 4(a) shows the configuration after acquiring the lock on bucket B_k and before moving keys K_1, K_2, and K_3 to the new hash level. Figure 4(b) illustrates key K_1 being moved to bucket B_m, while keys K_2 and K_3 are moved to bucket B_n in the new hash level. Once all keys have been moved, B_k is updated to reference the new hash level.

Finally, the deletion operation also begins by acquiring the lock for the bucket and searching for the target key K to be deleted. If K is found in the dynamic array, the last element of the array is copied into K's position, and the element count is decremented by one. It is important to note that the dynamic array is not deallocated, even when the number of stored elements reaches zero.

3.3 Optimizations

As mentioned earlier, vertical expansion is triggered when the total number of elements recorded in the hash header exceeds a predefined load factor. Since this count can change with every insertion or deletion, frequent updates to the

shared counter may cause significant contention under high thread concurrency. To reduce this overhead, each thread maintains a local counter to track its own successful insertions and deletions, updating the shared counter only after a fixed number of local operations. We refer to this optimization as *delayed updates*.

Moreover, for vertical expansion, we adopt a double-buffering strategy in which a new hash map is built in parallel and the buckets are migrated one by one. To accelerate this process, we implement a form of incremental rehashing where non-expanding threads assist by relocating individual buckets as they access them. We refer to this optimization as *cooperative expansion*.

4 Algorithms

We now present the algorithms that detail the core mechanisms of our two approaches, which we fully implemented from scratch. We begin with Alg. 1, which provides the pseudo-code for inserting a given (K, V) pair into a hash map H. Briefly, the $InsertOnHash()$ algorithm begins by computing the hash of key K to determine the appropriate bucket B within hash level H (line 1). It then attempts to acquire exclusive access to bucket B (line 2) and reads the current reference R stored at that location (line 3). If R indicates that B has already been expanded into a second hash level, the algorithm recursively invokes itself on that new hash level (lines 4–7).

Algorithm 1 *InsertOnHash(hash H, key K, Value V)*

```
 1: B ← Bucket(H, Hash(K, Size(H)))
 2: Lock(Mutex(B))
 3: R ← EntryRef(B)
 4: if IsHashRef(R) then   // R refers to a second hash level
 5:   nextH ← UnmaskHashRef(R)
 6:   Unlock(Mutex(B))
 7:   return InsertOnHash(nextH, K, V)
 8: else if IsHashExpanding(H) then   // cooperative expansion
 9:   nextH ← NextHash(H)
10:   AdjustBucket(B, nextH)
11:   EntryRef(B) ← MaskAsHashRef(nextH)
12:   Unlock(Mutex(B))
13:   return InsertOnHash(nextH, K, V)
14: else
15:   N ← InsertOnBucket(B, K, V)
16:   Unlock(Mutex(B))
17:   return N
```

Otherwise, the algorithm checks whether the hash level H is currently being expanded by another thread to a new hash level, $nextH$. If so, the current thread assists by expanding the current bucket B into $nextH$ using the $AdjustBucket()$ procedure (lines 8–13). After the expansion, it updates B to reference $nextH$ and recursively calls itself on the new hash level. The $MaskAsHashRef()$ procedure (line 11) applies a bitmask to mark a reference as pointing to a hash structure,

while the corresponding $UnmaskHashRef()$ procedure (line 5) removes this marker to retrieve the original reference.

If no expansion has been performed or is in progress, the algorithm proceeds to safely insert (K, V) into bucket B. This is achieved by invoking the auxiliary procedure $InsertOnBucket()$, which returns the updated number of key-value pairs in the bucket upon a successful insertion, or 0 if insertion fails (lines 15–17).

The $InsertOnBucket()$ procedure, shown in Alg. 2, assumes that the separate chaining mechanism uses linked lists. Due to space limitations, the array-based version is omitted; however, it is expected that the reader can easily grasp it. Conversely, for the delete procedure in Alg. 3, we provide the pseudo-code for the array-based implementation and omit the version based on linked lists.

Algorithm 2 *InsertOnBucket(bucket B, key K, Value V)* // for linked links

```
1: S ← 0    // size of the bucket chain
2: if EntryRef(B) = NULL then    // bucket is empty
3:   EntryRef(B) ← AllocNewNode(K, V)
4: else
5:   R ← EntryRef(B)
6:   repeat
7:     if Key(R) = K ∧ Value(R) = V then
8:       return 0    // the pair (K,V) was found
9:     else
10:      S ← S + 1
11:      Prev ← R
12:      R ← NextRef(R)
13:  until R ≠ NULL
14:  NextRef(Prev) ← AllocNewNode(K, V)
15: return S + 1
```

The $InsertOnBucket()$ algorithm begins by initializing a counter S to track the number of key-value pairs in the bucket (line 1). If the bucket is empty, a new node representing the pair (K, V) is allocated and initialized (lines 2–3). Otherwise, the algorithm enters a search phase, traversing the chain to locate the pair (K, V) (lines 5–13). If the pair is found, the procedure returns 0 to indicate that no insertion was performed (lines 7–8). If the pair is not present, a new node is allocated and appended to the end of the chain (line 14). Finally, the updated number of key-value pairs in the bucket is returned (line 15).

The deletion algorithm follows a structure similar to that of insertion. The $DeleteOnHash()$ algorithm begins by checking whether a second hash level exists or if it can assist in expanding the current bucket entry. In either case, as with insertion, it recursively calls itself on the next hash level. If no expansion has occurred or is in progress, the algorithm proceeds to safely delete the given (K, V) pair from bucket B by invoking the $DeleteOnBucket()$ procedure.

The $DeleteOnBucket()$ algorithm, shown in Alg. 3, assumes that separate chaining uses dynamic arrays. It starts by checking whether the given bucket B is empty, returning false if it is. Otherwise, it enters search mode, scanning the array A referenced by B for the target pair (K, V) (lines 4–10). If the pair is found at position $A[i]$, the array size S is decremented by one; the last element in

the array is moved to position $A[i]$, overwriting the target pair; and the updated size S is stored in B. The algorithm then returns true to indicate a successful deletion. If the pair is not found, it returns false (line 11).

Algorithm 3 *DeleteOnBucket(bucket B, key K, value V)* // for dynamic arrays

```
1: if EntryRef(B) = NULL then   // bucket is empty
2:   return False
3: else
4:   (A, S) ← EntryRef(B)
5:   for i = 0 to S − 1 do
6:     if Key(A[i]) = K ∧ Value(A[i]) = V then
7:       S ← S − 1
8:       A[i] ← A[S]
9:       EntryRef(B) ← (_, S)
10:      return True
11:  return False
```

Finally, Alg. 4 presents the pseudo-code for the vertical expansion procedure applied to a given hash H. Recall that vertical expansion is triggered after a successful insertion when both of the following conditions are satisfied: (i) the number of nodes in a chain reaches a specified threshold; and (ii) the total number of elements recorded in the hash header exceeds a predefined load factor.

Algorithm 4 *VerticalExpansion(hash H)*

```
1: if IsHashExpanding(H) then   // check for ongoing vertical expansion on H
2:   return
3: if TryLock(Mutex(H)) then   // try do expansion
4:   if IsHashExpanding(H) then   // recheck for ongoing vertical expansion on H
5:     Unlock(Mutex(H))
6:     return
7:   S ← Size(H)
8:   NextHash(H) ← AllocNewHash(2 × S)
9:   IsHashExpanding(H) ← True   // mark as ongoing vertical expansion on H
10:  Unlock(Mutex(H))
11:  nextH ← NextHash(H)
12:  for i = 0 to S − 1 do
13:    B ← Bucket(H, i)
14:    Lock(Mutex(B))
15:    R ← EntryRef(B)
16:    if not IsHashRef(R) then   // not expanded yet
17:      AdjustBucket(B, nextH)
18:      EntryRef(B) ← MaskAsHashRef(nextH)
19:    Unlock(Mutex(B))
```

To ensure that only one thread performs the hash expansion operation for a given hash H, the $VerticalExpansion()$ algorithm begins by checking whether an expansion is already in progress (line 1). If not, it attempts to acquire exclusive access to H (line 2). After acquiring the lock, it rechecks whether an expansion has started in the meantime and aborts if that is the case (lines 4–6). Note that if the call to $TryLock()$ fails, the thread proceeds without blocking and will

retry the operation in a subsequent attempt. If access is successfully granted, the algorithm proceeds to allocate a new hash with double the size of H, marks H as being in expansion, and then releases the lock (lines 7–10). The algorithm then iterates over the buckets of H, expanding each bucket B that has not yet been processed into the new hash $nextH$ using the *AdjustBucket*() procedure, and updates B to reference $nextH$ (lines 11–19).

We conclude with Alg. 5, which presents the *AdjustBucket*() procedure for dynamic arrays. The algorithm iterates over the elements of the array A referenced by the given bucket B, re-inserting each element into the next-level hash H. Once all elements have been reinserted, the current array A is deallocated.

Algorithm 5 *AdjustBucket(bucket B, hash H)* // for dynamic arrays

1: $(A, S) \leftarrow EntryRef(B)$
2: **for** $i = 0$ **to** $S - 1$ **do**
3: $InsertOnHash(H, key(A[i]), Value(A[i]))$
4: $FreeArray(A)$
5: **return**

5 Experimental Results

The experimental environment was based on a NUMA architecture with an Intel Core i9-10920X processor with 12 physical cores (24 hyperthreads) running at 3.50GHz. The system included 384KiB L1d + L1i (2 x 12 instances), 12MiB L2 (12 instances), 19.3MiB L3 (1 instance), and 251GB of main memory. It ran the Linux kernel 6.1.140 with GLIBC 2.36 (for the POSIX threads). All programs were compiled with GCC 13.3.0 and linked with the jemalloc memory allocator (version 5.3) [6]. To quantify the relevant metrics, particularly energy and cache performance trade-offs of both approaches, we used Linux's profiling tools with performance counters [1] powered by Intel's Running Average Power Limit (RAPL) interface [11].

5.1 Methodology

To evaluate the performance and energy behavior of separate chaining mechanisms in concurrent hash maps, we designed a set of benchmarks that explores various configurations using 1, 2, 4, 8, 12, 16, and 20 threads. Each benchmark was executed with a fixed workload of 8,388,608 operations and every configuration was repeated 10 times. We measured execution time, throughput (operations per second), energy consumption (via Intel RAPL), and, for cache behavior using *perf*, we collected statistics on *cache-references*, *cache-misses*, *L1-dcache-load-misses*, and *LastLevelCache-load-misses*. To ensure fairness, preparation steps, such as populating the hash map prior to lookup or delete operations, were excluded from execution time and energy measurements. Both implementations were carefully aligned to 64-byte cache lines to avoid alignment faults, which we verified using *perf*. Additionally, global synchronization flags were used to ensure that all threads started simultaneously.

5.2 Performance Analysis

To simulate realistic usage patterns, we adopted a methodology inspired by the YCSB benchmarking framework [4], where each workload scenario is defined by a specific combination of operation ratios. Specifically, we evaluated both designs under four representative workloads: (i) 100% insertions; (ii) 100% lookups; (iii) 80% lookups combined with 10% insertions and 10% deletions; and (iv) 60% lookups combined with 20% insertions and 20% deletions. These combinations reflect a spectrum of access patterns ranging from write-heavy to read-dominant workloads, and align with widely adopted experimental practices in the literature [17, 9, 14]. Each hash map was further evaluated under load factors of 3 and 5, resulting in four variants: LL-3 and LL-5 for the linked list approach, and DA-3 and DA-5 for the dynamic array approach.

Figure 5 presents the results for the 100% insertions benchmark, which is useful for analyzing the impact of both horizontal and vertical expansion. Throughput scales well with the number of threads for all variants, but dynamic arrays grow more sharply. In terms of energy consumption, all variants show a steady upward trend, but LL-3 reaches a peak at 12 threads before dropping, indicating non-linear performance/power behavior. This can be attributed to LL-3's poor cache performance at 12 threads, as confirmed by the remaining figures, which focus on cache-related metrics.[1] Overall, both linked list variants consistently exhibit higher cache miss rates across all cache levels compared to their dynamic array counterparts.

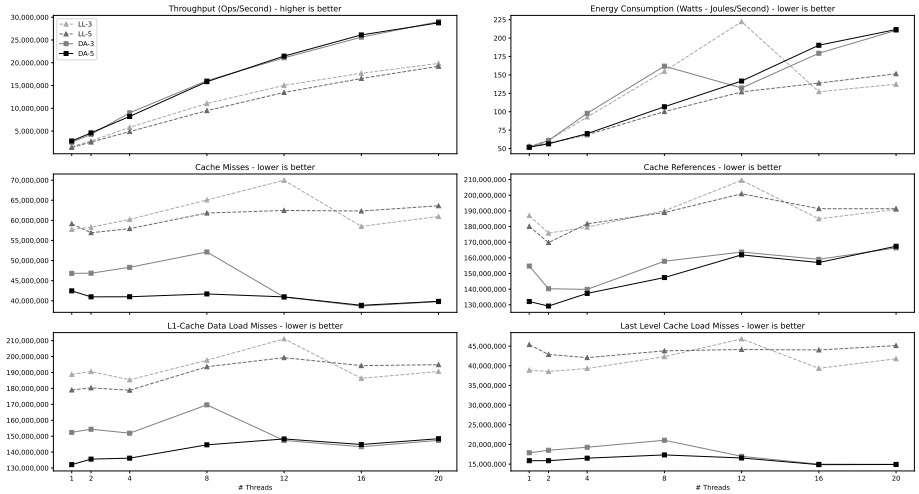

Fig. 5. Lists vs Arrays - 100% Insertions

Figure 6 presents the results for the 100% lookups benchmark. The highest throughput is achieved by DA-3, closely followed by DA-5. The linked list variants consistently performs worse. As expected, dynamic arrays benefit from

[1] This effect may be related to the underlying CPU architecture and requires further study. Note that the host CPU has 12 physical cores and supports 24 hyperthreads.

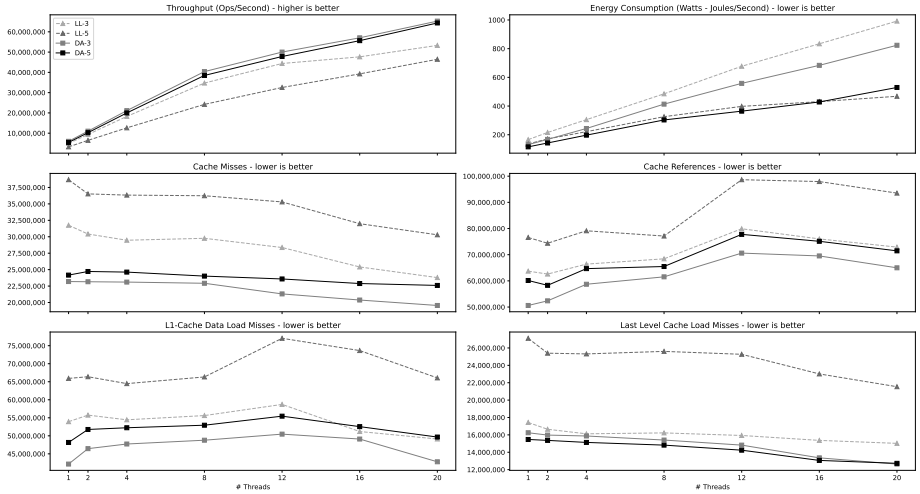

Fig. 6. Lists vs Arrays - 100% Lookups

better cache locality, especially under read-heavy workloads like this one. On the other hand, linked lists suffer from pointer chasing, which leads to more cache references, poor spatial locality, and consequently higher cache miss rates. One can observe that LL-5 performs significantly worse than all other variants. Interestingly, LL-3's cache performance approaches that of DA-5 when using 20 threads, suggesting that in this particular benchmark, the negative impact of pointer chasing in LL-3 becomes negligible at high thread counts.

Next, Figures 7 and 8 show how results evolve as the proportion of lookup operations decreases and the share of insertion and deletion operations increases.

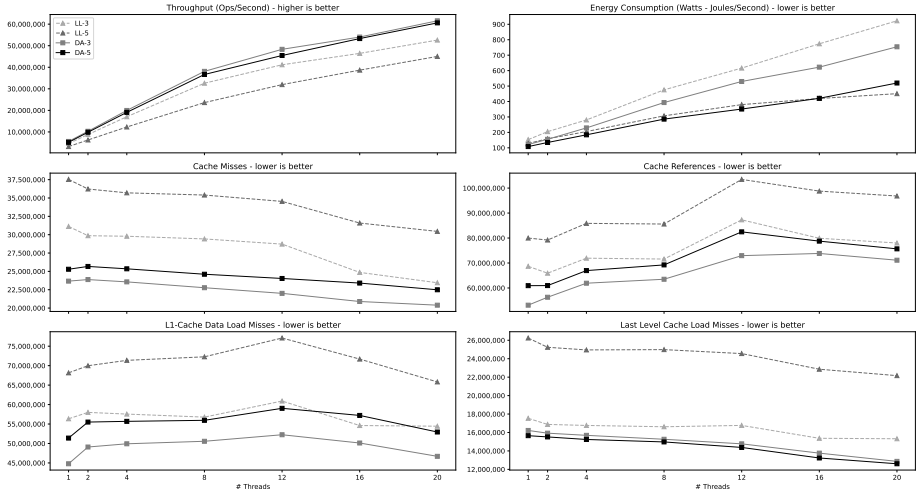

Fig. 7. Lists vs Arrays - 80% Lookups + 10% Insertions/Deletions

Across both workloads, dynamic arrays consistently outperform linked lists, especially as the number of threads increases. Dynamic arrays exhibit better

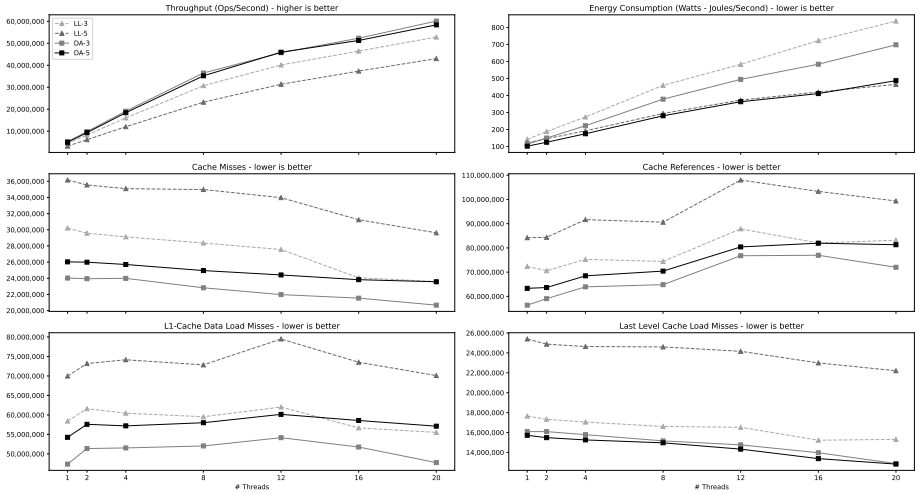

Fig. 8. Lists vs Arrays - 60% Lookups + 20% Insertions/Deletions

scalability, whereas linked lists, particularly LL-5, lag behind. In terms of energy consumption, dynamic arrays generally consume less energy, with the gap increasing at higher thread counts. This suggests that the better spatial locality of arrays reduces both memory and energy consumption, even when updates occur frequently. Analyzing cache behavior, both figures show that dynamic arrays have lower cache misses, both L1 data and last-level, compared to linked lists. This advantage is especially visible in Fig. 7, where the high proportion of lookup operations amplifies the overhead of pointer chasing in linked lists. In particular, LL-5 has more cache references and misses, likely due to longer chains and increased memory traversal. In contrast, dynamic arrays store elements contiguously, reducing memory indirection and improving cache efficiency.

6 Conclusions & Further Work

This work offers a comprehensive comparison of linked lists and dynamic arrays for separate chaining in multithreaded lock-based hash maps, evaluating them in terms of throughput, multi-level cache performance, and energy efficiency.

Experimental results consistently show that dynamic arrays offer more predictable memory access patterns and lower energy consumption in multithreaded scenarios. Dynamic arrays achieve higher throughput across all thread counts, with better scalability and reduced cache overhead. This advantage stems from improved spatial locality, which minimizes L1 and last-level cache misses and enhances memory efficiency. In contrast, linked lists suffer from pointer-chasing and fragmentation, particularly under high load factors.

As further work, we plan to extend our study by investigating how different synchronization mechanisms, such as read-write locks, lock-free designs, and lock-free locks [3], impact the performance of the dynamic arrays approach.

Acknowledgments

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2023 (https://doi.org/10.54499/UID/50014/2023). Ana Castro was supported by a BII grant from INESC TEC.

References

1. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org (2024), accessed June 2025
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman, USA (2006)
3. Ben-David, N., Blelloch, G.E., Wei, Y.: Lock-free locks revisited. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 278–293. ACM (2022)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: ACM Symposium on Cloud Computing. p. 143–154. ACM (2010)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2009)
6. Evans, J.: A scalable concurrent malloc (3) implementation for FreeBSD. In: BSDCan Conference (2006)
7. Farach-Colton, M., Krapivin, A., Kuszmaul, W.: Optimal Bounds for Open Addressing Without Reordering. In: Symposium on Foundations of Computer Science. pp. 594–605 (2024)
8. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann (2012)
9. Hu, D., Chen, Z., Wu, J., Sun, J., Chen, H.: Persistent memory hash indexes: an experimental evaluation. VLDB Endowment **14**(5), 785–798 (2021)
10. Katsaragakis, M., Baloukas, C., Papadopoulos, L., Kantere, V., Catthoor, F., Soudris, D.: Energy Consumption Evaluation of Optane DC Persistent Memory for Indexing Data Structures. In: IEEE International Conference on High Performance Computing, Data, and Analytics. pp. 75–84 (2022)
11. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: RAPL in Action: Experiences in Using RAPL for Power Measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems **3**(2) (2018)
12. Knuth, D.E.: The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison-Wesley Longman (1998)
13. Lehman, T.J., Carey, M.J.: A Study of Index Structures for Main Memory Database Management Systems. In: International Conference on Very Large Data Bases. p. 294–303. Morgan Kaufmann (1986)
14. Maier, T., Sanders, P., Dementiev, R.: Concurrent Hash Tables: Fast and General(?)! ACM Transactions on Parallel Computing **5**(4), 1–32 (2019)
15. Maurer, W.D., Lewis, T.G.: Hash Table Methods. ACM Computing Surveys **7**(1), 5–19 (1975)
16. Tenenbaum, A.M., Langsam, Y., Augenstein, M.J.: Data Structures Using C. Prentice Hall (1990)

17. Wang, C., Hu, J., Yang, T., Liang, Y., Yang, M.: SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory. In: USENIX Symposium on Operating Systems Design and Implementation. pp. 479–495. USENIX Association (2023)
18. Xu, S., Liu, D.: Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study. International Journal of Networked and Distributed Computing **3**(1), 60–68 (2015)