*mathematics*

MDPI

*Article*

# Performance Evaluation of Separate Chaining for Concurrent Hash Maps

**Ana Castro *, Miguel Areias * and Ricardo Rocha**

CRACS & INESC TEC and Faculty of Sciences, University of Porto, Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal; ricroc@dcc.fc.up.pt

* Correspondence: ana-castro@dcc.fc.up.pt (A.C.); miguel-areias@dcc.fc.up.pt (M.A.)

**Abstract**

Hash maps are a widely used and efficient data structure for storing and accessing data organized as key-value pairs. Multithreading with hash maps refers to the ability to concurrently execute multiple lookup, insert, and delete operations, such that each operation runs independently while sharing the underlying data structure. 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 evaluation 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.

**Keywords:** hash maps; concurrency; performance evaluation

**MSC:** 68W10

Academic Editor: Ivan Lirkov

Received: 24 July 2025
Revised: 19 August 2025
Accepted: 30 August 2025
Published: 2 September 2025

**Citation:** Castro, A.; Areias, M.; Rocha, R. Performance Evaluation of Separate Chaining for Concurrent Hash Maps. *Mathematics* **2025**, *13*, 2820. https://doi.org/10.3390/math13172820



## 1. Introduction

Hash maps [1] 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 [3], and database indexing mechanisms [4]. A key aspect of hash map design is the management of collisions. *Separate chaining* [5,6] is a widely adopted strategy for collision resolution, typically implemented using either *linked lists* or *dynamic arrays*. Both approaches can be elastic enough to accommodate an unbounded number of collisions.

Concurrent hash maps aim to retain the advantages of their non-concurrent counterparts while ensuring correctness and high performance in multithreaded environments [7]. In particular, a good collision management strategy is vital under concurrent hash map designs [8]. Traditional synchronization approaches rely on varying levels of lock granularity, ranging from coarse-grained to fine-grained locking. The choice of synchronization has a significant impact on performance, particularly under high contention and workloads with frequent insertion, deletion, and lookup operations. 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. In particular, recent studies have started to explore the energy profiles of data structures under concurrent workloads, aiming to better understand how the irregular data accesses in concurrent hash maps can significantly impact the energy consumption of memory operations [9]. However, to the best of our knowledge, there is no comprehensive study that performs an in-depth comparison between separate chaining mechanisms, linked list-based and dynamic array-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 energy, cache behavior, contention under concurrent access, and resizing overhead—remain largely unexplored.

The remainder of this paper is organized as follows. We begin by introducing the necessary related work and background, and provide an overview of the two separate chaining approaches that were used in this work. 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) [10,11] and Linux performance counters [12] 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

Green software has emerged as an important area of research, as modern systems increasingly prioritize not only performance but also energy efficiency [13–15]. In the context of data structures, energy evaluation has been approached from both theoretical and empirical perspectives. On the theoretical side, energy-aware algorithmic cost models formalize how computation, memory access, and communication contribute to energy complexity [16]. On the practical side, studies on performance and energy often rely on specialized tools and libraries, such as Linux's profiling support with hardware performance counters [12] and Intel's Running Average Power Limit (RAPL) interface [10]. Both provide good estimations with sufficiently low overhead, making them suitable for profiling concurrent (multithreaded) environments [11].

Concurrent hash map data structures represent a significant area of research due to the challenges of maintaining both performance and consistency under concurrent access. For example, Maier et al. [8] performed an extensive experimental study comparing their implementations against six widely used concurrent hash maps. Other studies have explored synchronization strategies to address these challenges, including lock-based, lock-free, and wait-free approaches. Notable examples include the following: (i) the wait-free resizable hash map by Fatourou et al. [17]; (ii) DHash, a dynamic hash map enabling on-the-fly scaling by Wang et al. [18]; (iii) Ctries, a tree-based structure with efficient updates and snapshot support, proposed by Prokopec et al. [19]; (iv) the split-ordered lists introduced by Shalev and Shavit, which enables lock-free incremental resizing of hash maps [20]; and (v) Malakhov's description of Intel's concurrent hash map implementation in the Threading Building Blocks (TBB) library [21].

In addition, specialized concurrent hash map designs target specific real-world applications. Examples include the following: (i) concurrent cuckoo hashing optimized for networking scenarios [22]; (ii) concurrent hash maps designed for data stream processing [23]; and (iii) Red Hat's fast, dynamically resizable concurrent hash map, which emphasizes both safety and performance [24].

One of the main challenges in any hash map implementation is the management of collisions, which occur when different keys are mapped to the same bucket. 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. Common strategies for collision resolution include *open addressing* and *closed addressing*.

In open addressing, values are stored in the hash map itself and locations within the array structure are systematically probed to resolve collisions [5,6]. A variety of open addressing schemes—such as linear probing, quadratic probing, cuckoo hashing, hopscotch hashing—offer different trade-offs in terms of performance and cache locality. While open addressing is attractive in terms of memory overhead, it can suffer from clustering, which requires careful tuning of the *load factor* (the ratio of stored elements to available buckets) to maintain performance [6,25–27].

In contrast, closed addressing approaches, such as *separate chaining*, resolve collisions by associating each bucket with a secondary data structure that stores multiple values corresponding to colliding keys. Separate chaining remains effective under moderate-to-high load factors, but it introduces additional memory overhead and may degrade performance when chains grow long [6].

When implementing separate chaining, two common data structures are *linked lists* and *dynamic arrays*. 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. This issue is well documented, and one can find multiple attempts in the literature to mitigate it. For example, Zobel and Askitis explored contiguous-node and array-based collision chains for strings, demonstrating substantial gains in both time and cache-miss rates [28]. They also developed advanced hybrid data structures, such as HAT-tries, which blend hash buckets and arrays for caching efficiency [29].

Dynamic arrays also 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 [30,31].

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 the vertical and horizontal expansions. This requires careful synchronization to prevent other threads from reading or writing while buckets are being expanded. 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 [8,20,21,32].

## 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.

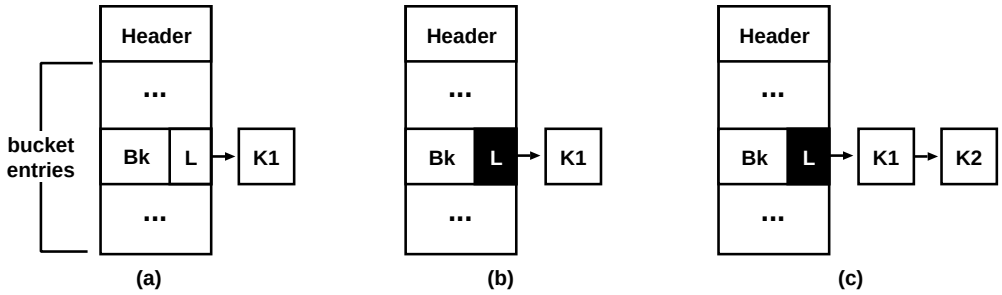


**Figure 1.** Concurrent insertion with linked lists.

Figure 1a 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 1b 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 1c 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.

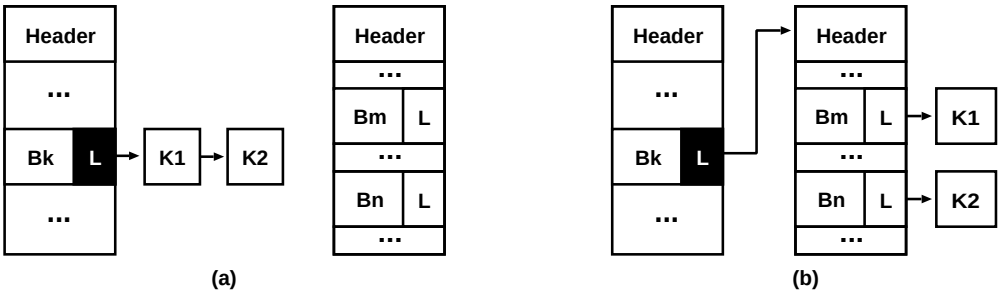


**Figure 2.** Concurrent vertical expansion with linked lists.

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 2a 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 2b 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 Figure 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.

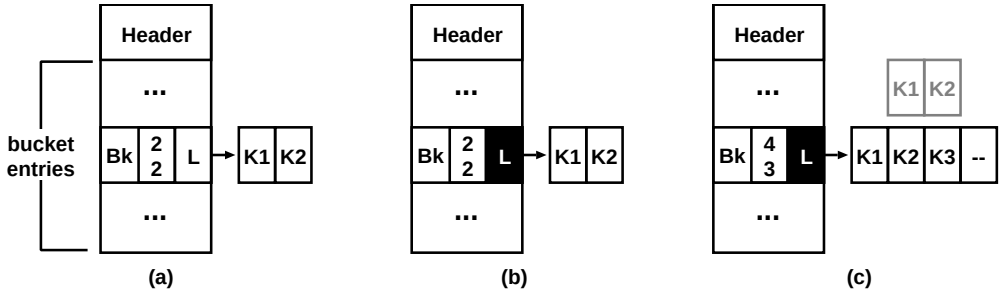


**Figure 3.** Concurrent insertion with dynamic arrays.

Figure 3a 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 3b shows the hash map configuration just before inserting a new key $K_3$, which requires acquiring the lock for that bucket. Finally, Figure 3c 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.

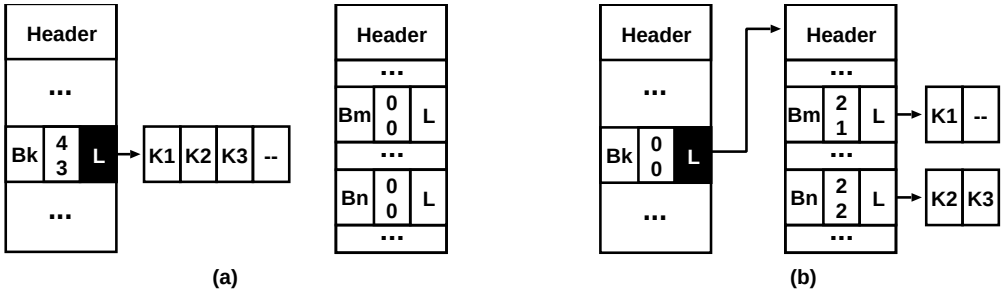


**Figure 4.** Concurrent vertical expansion with dynamic arrays.

Figure 4a 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 4b 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*. Note that this optimization is triggered specifically during the insert operation, when a thread accesses a bucket in a hash map that is currently expanding. At that point, the thread migrates any pre-existing elements into the new hash map before completing its initial task. Through empirical experimentation, we observed that the time required to migrate a bucket during an insert operation is negligible compared to the positive impact on performance obtained by this optimization. To mitigate contention between the expanding thread and the helper threads, our approach uses power-of-two hash sizes along with a simple bit-wise modulo hashing function, such that elements from a given bucket in the old hash can only relocate to two possible positions in the new hash. In practice, this means that a thread expanding a bucket will be the only thread using the two relocation options, effectively eliminating competition among threads for the same buckets in the new hash.

## 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 Algorithm 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).

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.

**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
18: end if
```

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 Algorithm 2, assumes that the separate chaining mechanism uses linked lists. To keep the discussion concise, the array-based version is omitted; however, it is expected that the reader can easily grasp it. Conversely, for the delete procedure in Algorithm 3, we provide the pseudo-code for the array-based implementation and omit the version based on linked lists.

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 5–14). 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 15). Finally, the updated number of key-value pairs in the bucket is returned (line 17).

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 Algorithm 3, assumes that separate chaining uses dynamic arrays. It starts by checking whether the given bucket $B$ is empty,

returning false if it is (lines 1–2). Otherwise, it enters search mode, scanning the array $A$ referenced by $B$ for the target pair $(K, V)$ (lines 4–12). 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 (line 10). If the pair is not found, it returns false (line 13).

**Algorithm 2** *InsertOnBucket(bucket B, key K, value V)* // for linked lists

1: $S \leftarrow 0$ ▷ Size of the bucket chain
2: **if** $EntryRef(B) =$ NULL **then** ▷ Bucket is empty
3: $\quad EntryRef(B) \leftarrow AllocNewNode(K, V)$
4: **else**
5: $\quad R \leftarrow EntryRef(B)$
6: $\quad$ **repeat**
7: $\quad\quad$ **if** $Key(R) = K \wedge Value(R) = V$ **then**
8: $\quad\quad\quad$ **return** 0 ▷ The pair (K, V) was found
9: $\quad\quad$ **else**
10: $\quad\quad\quad S \leftarrow S + 1$
11: $\quad\quad\quad Prev \leftarrow R$
12: $\quad\quad\quad R \leftarrow NextRef(R)$
13: $\quad\quad$ **end if**
14: $\quad$ **until** $R =$ NULL
15: $\quad NextRef(Prev) \leftarrow AllocNewNode(K, V)$
16: **end if**
17: **return** $S + 1$

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

1: **if** $EntryRef(B) =$ NULL **then** ▷ Bucket is empty
2: $\quad$ **return False**
3: **else**
4: $\quad (A, S) \leftarrow EntryRef(B)$
5: $\quad$ **for** $i = 0$ **to** $S - 1$ **do**
6: $\quad\quad$ **if** $Key(A[i]) = K \wedge Value(A[i]) = V$ **then**
7: $\quad\quad\quad S \leftarrow S - 1$
8: $\quad\quad\quad A[i] \leftarrow A[S]$
9: $\quad\quad\quad EntryRef(B) \leftarrow (\_, S)$
10: $\quad\quad\quad$ **return True**
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad$ **return False**
14: **end if**

Finally, Algorithm 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:  else if TryLock(Mutex(H)) then                      ▷ Try to do expansion
4:      if IsHashExpanding(H) then                      ▷ Recheck for ongoing expansion on H
5:          Unlock(Mutex(H))
6:          return
7:      end if
8:      S ← Size(H)
9:      NextHash(H) ← AllocNewHash(2 × S)
10:     IsHashExpanding(H) ← True                       ▷ Mark expansion as ongoing on H
11:     Unlock(Mutex(H))
12:     nextH ← NextHash(H)
13:     for i = 0 to S − 1 do
14:         B ← Bucket(H, i)
15:         Lock(Mutex(B))
16:         R ← EntryRef(B)
17:         if not IsHashRef(R) then                    ▷ Not expanded yet
18:             AdjustBucket(B, nextH)
19:             EntryRef(B) ← MaskAsHashRef(nextH)
20:         end if
21:         Unlock(Mutex(B))
22:     end for
23: end if
```

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 3). After acquiring the lock, it rechecks whether an expansion has started in the meantime and aborts if that is the case (lines 4–7). 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 8–11). 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 12–22).

We conclude with Algorithm 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 $H$ (lines 1–4). Once all elements have been reinserted, the current array $A$ is deallocated (line 5).

**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: $\quad InsertOnHash(H, \text{key}(A[i]), \text{Value}(A[i]))$
4: **end for**
5: $FreeArray(A)$
6: **return**

Finally, Figure 5 presents a consolidated view of the insertion process, illustrating how the algorithms *InsertOnHash()*, *InsertOnBucket()*, and *VerticalExpansion()* interact. The figure summarizes the sequence from identifying the correct hash level through recursive restarts, to detecting duplicates early in order to avoid unnecessary work, and to triggering vertical expansion when the structural conditions are met.

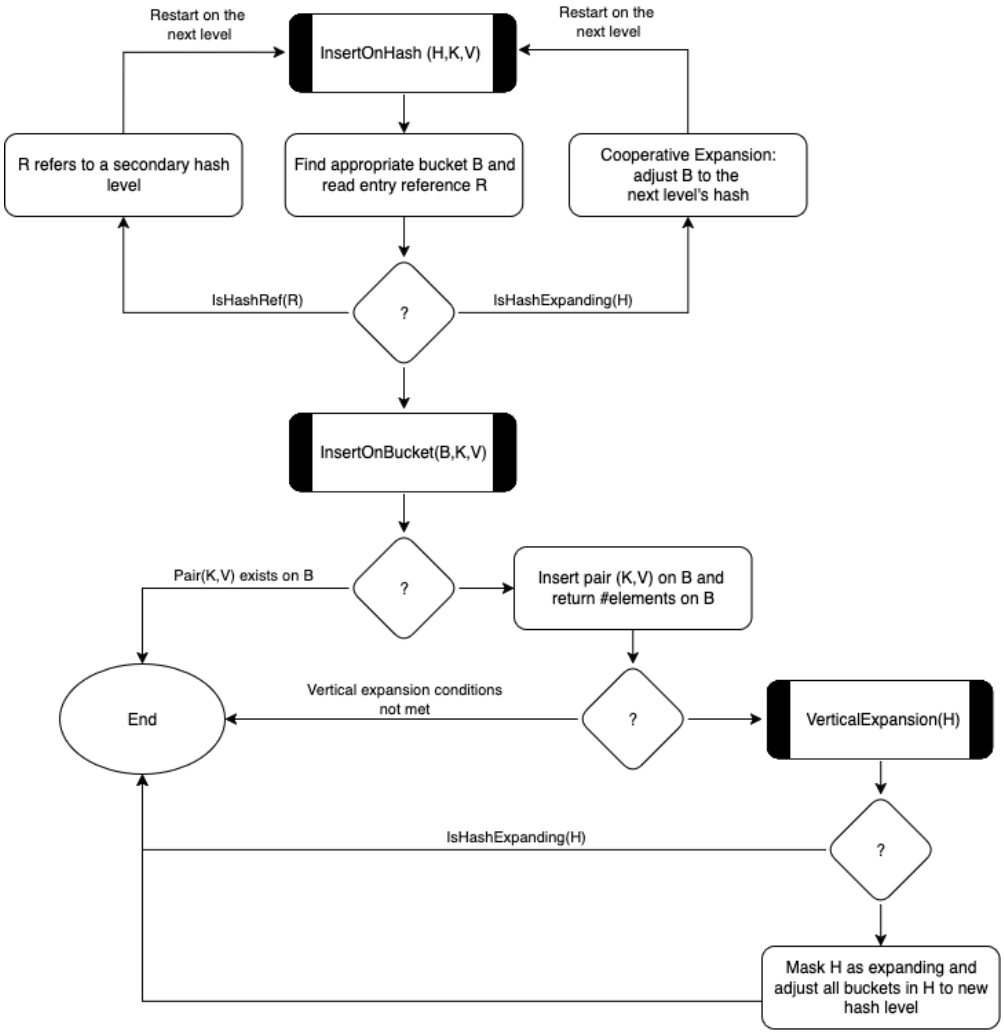


**Figure 5.** Execution flow of the insertion process.

## 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 3.50 GHz. The system included 384KiB L1d + L1i (2 × 12 instances), 12MiB L2 (12 instances), 19.3MiB L3 (1 instance), and 251 GB 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) [33]. To quantify the relevant metrics, particularly energy and cache performance trade-offs of both approaches, we used Linux's profiling tools with performance counters [12] powered by Intel's Running Average Power Limit (RAPL) interface [10,11].

### *5.1. Methodology*

To evaluate the performance and energy efficiency of separate chaining mechanisms in concurrent hash maps, we relied on an existing benchmarking tool, described in detail by Moreno et al. [34], which has been widely adopted in several recent works [35–37]. This tool enables the measurement of execution time by setting the following parameters: the number of threads to use, the total number of operations to execute, and the distribution of insert, delete, and lookup operations. To ensure that the corresponding key-value pairs exist in the hash map when required by the delete and lookup operations, the tool implements a setup stage in which such pairs are pre-inserted into the hash map before benchmarking begins.

To integrate our hash map with the benchmarking tool, we created a support a data structure that serves as an interface, enabling the tool to control execution and run the required operations. This support structure maintains a pointer to the current hash map instance, a lock to protect that pointer, and a header that includes an array of counters used for bookkeeping as part of the delayed updates optimization. Each thread is assigned a unique ID, which it uses to access its corresponding entry in the counters array without requiring locking. Each thread tracks (i) the total number of operations it has executed, (ii) the number of operations since its last header update, and (iii) the number of elements to add or subtract from the header when an update is eventually performed.

Although integration was fairly easy, using the total number of operations as an upper limit proved problematic with vertical expansion. Specifically, this caused running times to be dominated by a single thread performing the final expansion alone, effectively executing many operations serially near the end of the benchmark. As a result, increasing the number of threads resulted in performance degradation, as many threads would finish progressively earlier. To address these effects and improve fairness across threads, we implemented a global synchronization mechanism to ensure that all threads started and completed their operations as simultaneously as possible, minimizing discrepancies in thread start times and helping ensure a more uniform distribution of work.

The adopted solution proceeds as follows: after launching all threads and signaling them to begin execution via a shared synchronization flag, the main thread monitors progress by scanning the counters array. When the main thread detects that any thread has completed its assigned batch of operations, it resets the shared flag to signal all threads to stop, and then waits for their termination. Execution time is measured by recording timestamps immediately before setting the start flag and after signaling the stop condition. With both the total number of operations and elapsed time performed, throughput can be accurately computed. This approach allows performance measurement without either fixing a time limit or depending on the operation count, ensuring fairness and consistency, especially in insert-heavy workloads.

For our experiments, 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, repeating every configuration 10 times. We measured execution time, throughput in operations per second (throughput metrics may, in some cases, reflect the volume of data processed rather than the number of operations; in this work, the data we are using is always a key/value pair associated with an insertion, lookup, or delete operation), 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*. Both implementations were carefully aligned to 64-byte cache lines to avoid alignment faults, which we verified using *perf*.

Additionally, preparatory steps, such as the benchmark's setup stage, were excluded from execution time, energy consumption, and cache-related metrics. To exclude these readings, we performed auxiliary runs that measured these parameters just for the setup stage of each configuration tested. We could then calculate the average for each parameter for the setup stage and deduct it from the respective full benchmark run. While this method does not yield perfectly precise results, it was the most practical approach we could find given the current limitations of the *perf* tool, which does not natively support partial measurement of execution intervals.

### *5.2. Performance Evaluation*

To simulate realistic usage patterns, we adopted a methodology inspired by the YCSB benchmarking framework [38], where each workload scenario is defined by a specific combination of operation ratios. Specifically, we evaluated both designs under six representative workloads: (i) 100% insertions; (ii) 100% lookups; (iii) 80% lookups combined with 10% insertions and 10% deletions; (iv) 60% lookups combined with 20% insertions and 20% deletions; (v) 40% lookups combined with 30% insertions and 30% deletions; and (vi) 100% 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 [8,39,40]. Each hash map was further evaluated under load factors of 3, 5, and 7, resulting in six variants: LL-3, LL-5, and LL-7 for the linked list approach, and DA-3, DA-5, and DA-7 for the dynamic array approach. For all variants, the initial size of the hash map was set to $2^{14}$ (16,384) bucket entries, the local counter for the delayed updates optimization was fixed at 1000 operations before updating the shared counter, and the initial array size of the DA approaches (DA-3, DA-5, and DA-7) was fixed at four elements, ensuring a fair comparison between linked lists and dynamic arrays in terms of cache alignment.

Figure 6 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. LL-5 and LL-7 also show a change in the performance at 12 threads, but the energy consumption does not decrease as the remaining variants (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). Overall, all linked list variants consistently exhibit higher cache miss rates across all cache levels compared to their dynamic array counterparts.

Figure 7 presents the results for the 100% lookups benchmark. The highest throughput is achieved by the DA variants. The LL variants consistently perform worse. As expected, dynamic arrays benefit from 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-7 performs worse than all other variants. Interestingly, LL-3's cache performance approaches that of DA-5 and DA-7, 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.

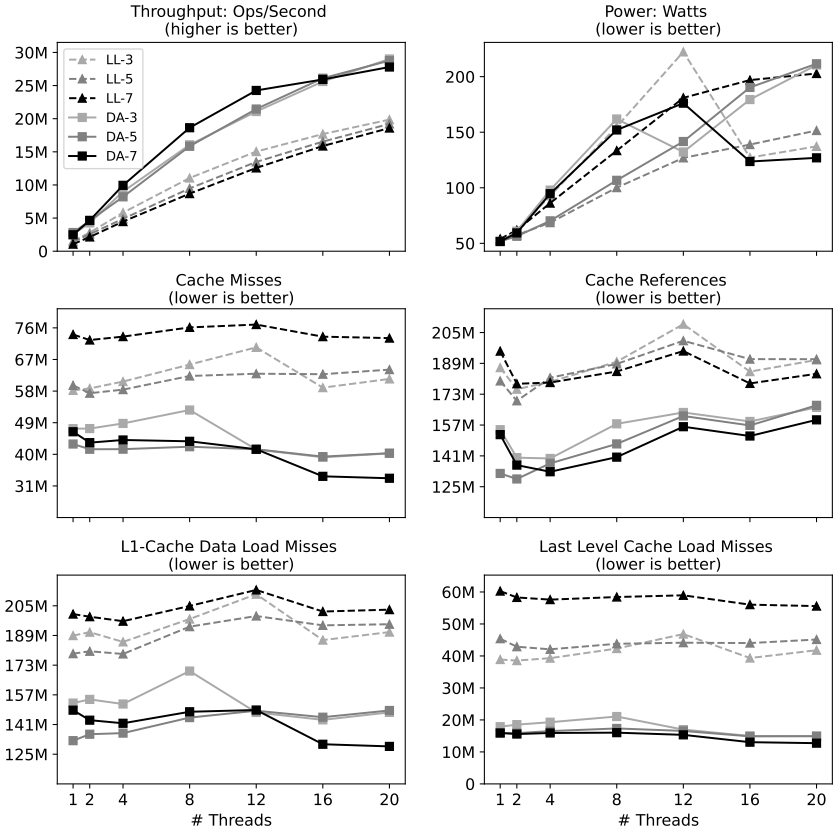


**Figure 6.** Lists vs. arrays—100% insertions.

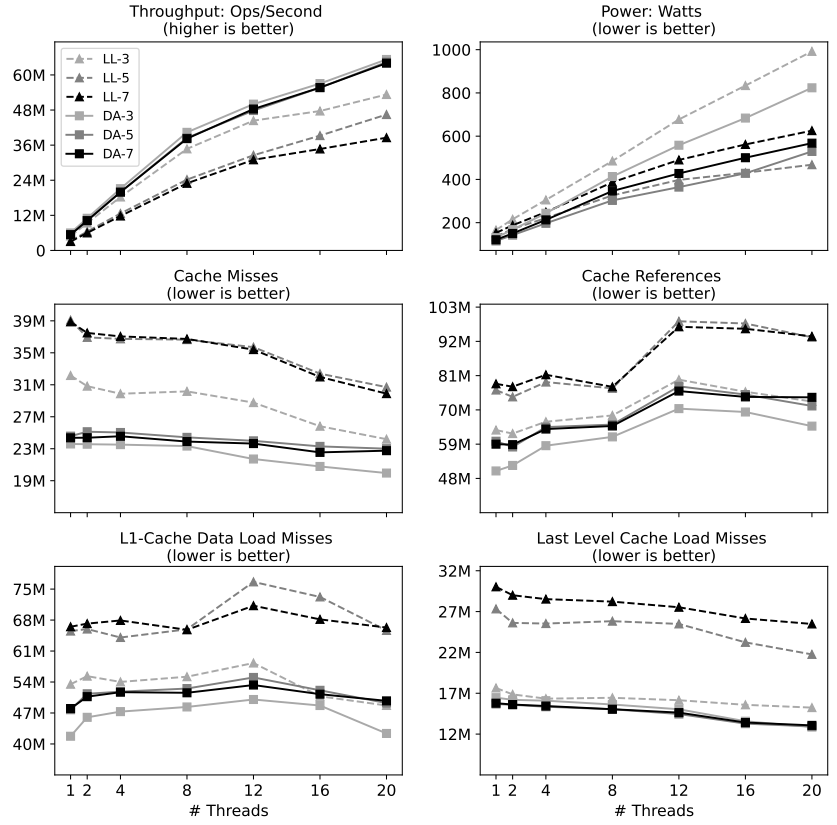


**Figure 7.** Lists vs. arrays—100% lookups.

Next, Figures 8–10 show how results evolve as the proportion of lookup operations decreases, and the share of insertion and deletion operations increases.

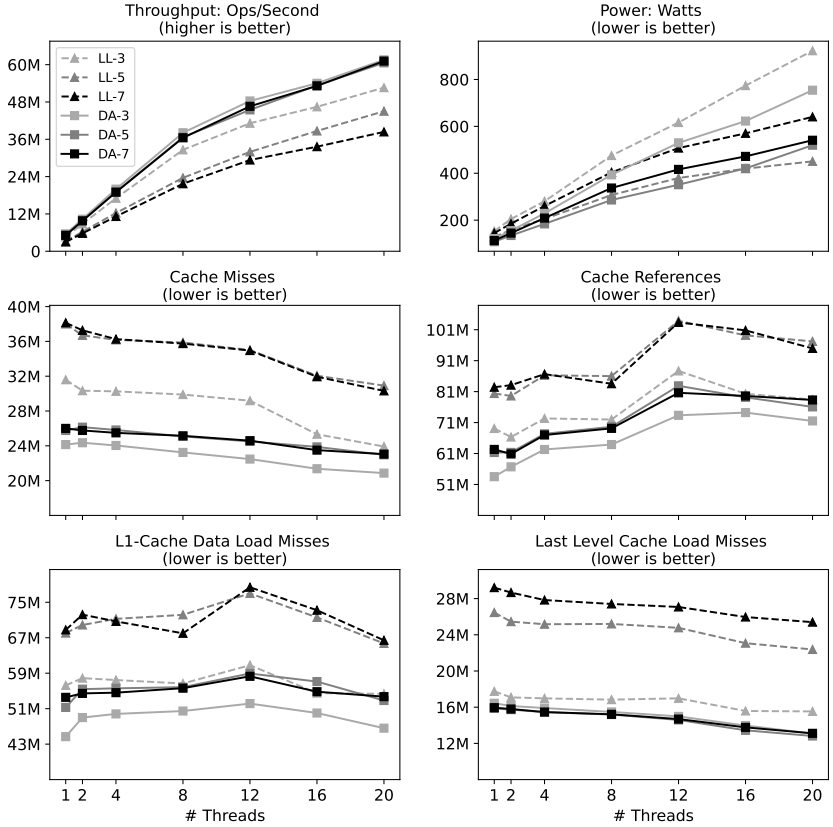


**Figure 8.** Lists vs. arrays—80% lookups + 10% insertions/deletions.

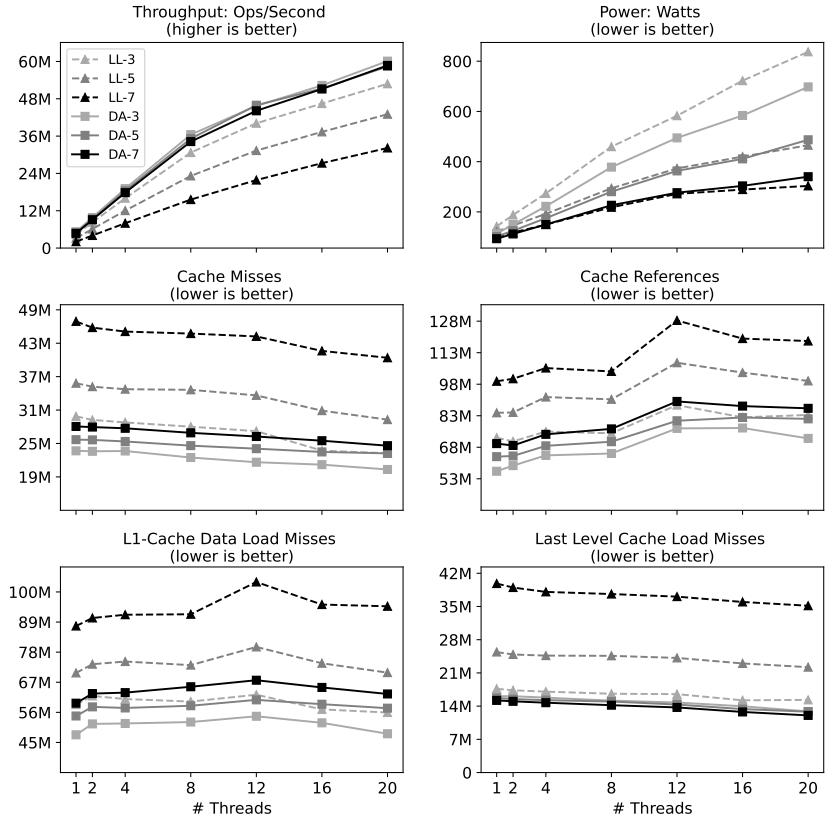


**Figure 9.** Lists vs. arrays—60% lookups + 20% insertions/deletions.

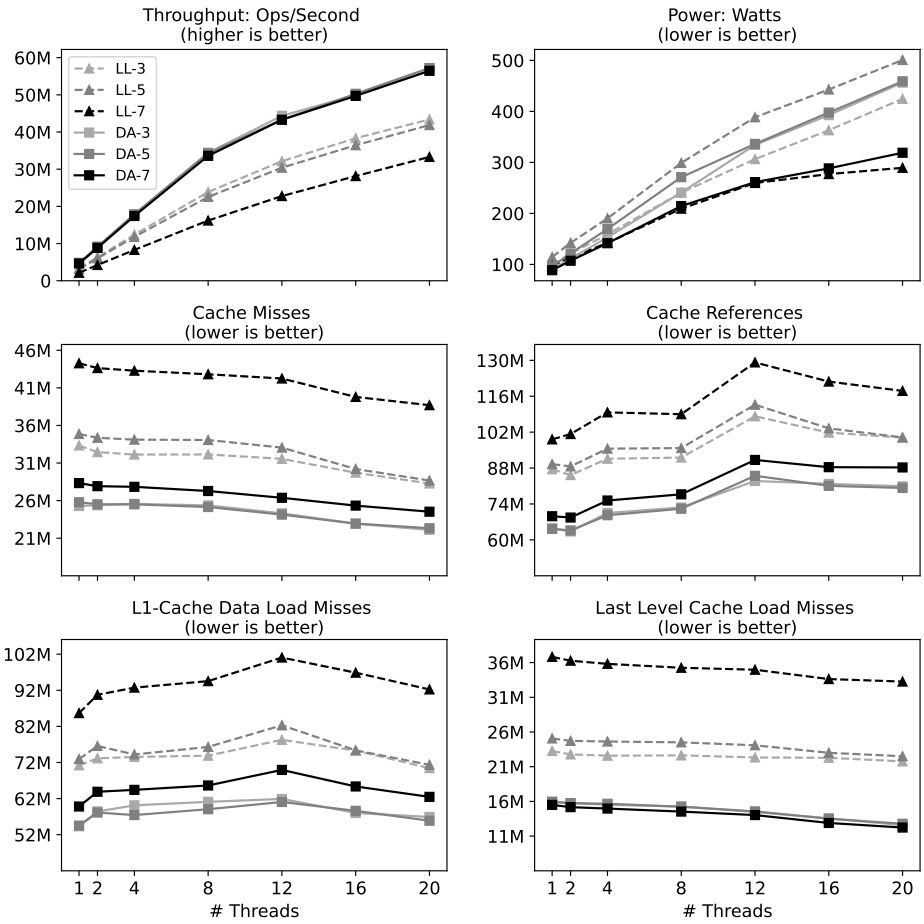


**Figure 10.** Lists vs. arrays—40% lookups + 30% insertions/deletions.

Across these workloads, dynamic arrays consistently outperform linked lists, especially as the number of threads increases. Dynamic arrays exhibit better scalability, whereas linked lists, particularly LL-7, 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 Figure 8, where the high proportion of lookup operations amplifies the overhead of pointer chasing in linked lists. In particular, LL-5 and LL-7 have 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.

Finally, Figure 11 presents the results for the 100% deleted benchmark. This workload is particularly interesting because it involves intensive writing activity. Each deletion operation typically triggers multiple write operations and does not include any structural expansion. In contrast, all other evaluated scenarios either include expansion (e.g., insert-heavy workloads) or consist solely of read operations, such as the 100% searches scenario. The absence of expansion in this case allows us to isolate and better understand the performance implications of write-heavy workloads independent of growth-related overhead.

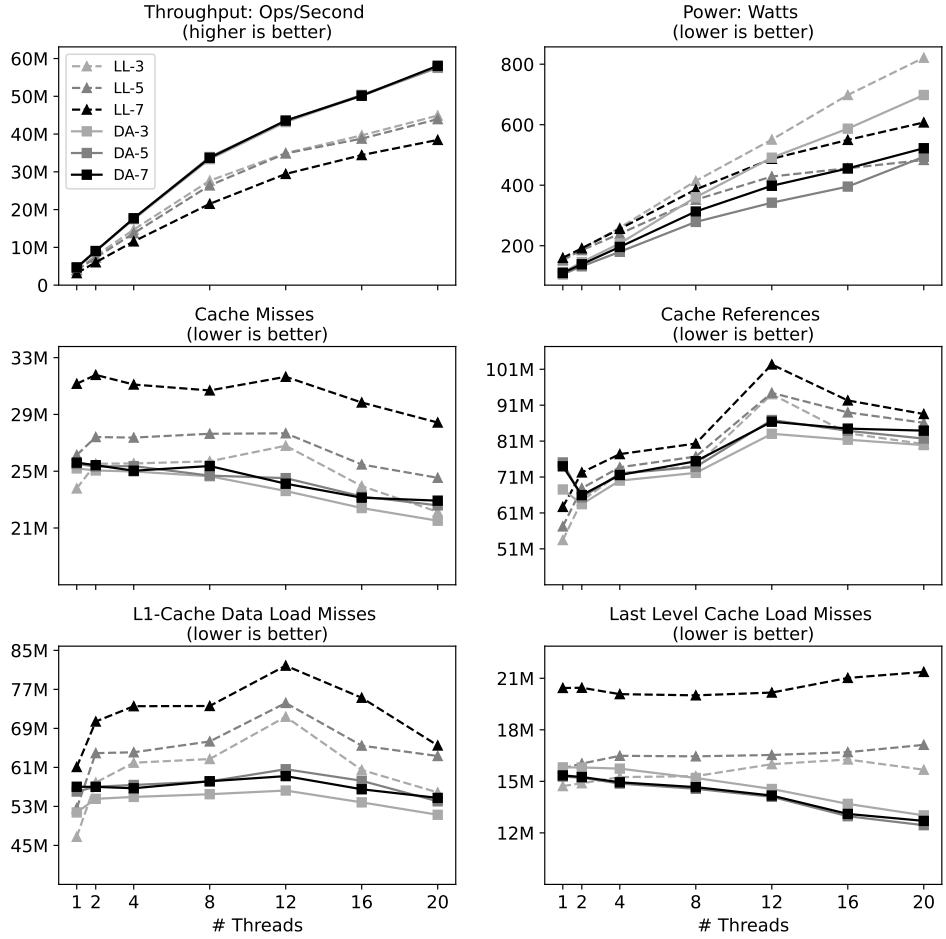


**Figure 11.** Lists vs. arrays—100% deletions.

In the 100% deletion benchmark, DA variants consistently outperform their LL counterparts in terms of throughput, especially under high thread counts. The performance gap becomes more pronounced with increasing concurrency, highlighting the benefits of a compact memory layout and reduced pointer traversal during write-intensive operations. Cache analysis further reveals additional insights. DA-based variants generate fewer cache references and incur a lower number of cache misses compared to LL variants, especially at higher load factors. Notably, LL-3 exhibits similar cache misses to that of the DA variants. Key divergence in cache performance between both variants actually occurs at the last-level cache, highlighting a clear advantage for DA-based variants, thus indicating that their more cache-friendly memory access patterns play a critical role in sustaining higher throughput under contention.

### *5.3. Discussion*

The performance evaluation across multiple workloads reveals clear differences between the two separate chaining strategies—linked lists (LLs) and dynamic arrays (DAs)—under varying load factors and thread counts. Across all benchmarks, DA variants consistently achieved higher throughput, particularly under high thread counts, whereas LL-based variants showed saturation or degradation due to the overhead of pointer-heavy traversal.

DA variants maintained stable and competitive throughput across increasing thread counts in nearly all workloads. This suggests that, for randomly distributed keys (as used in this study), the load factor does not significantly impact performance for DA-based chaining. In contrast, LL variants exhibited poor throughput with deeper chains, leading to worse performance due to increased traversal costs. Among LL variants, shorter

chains consistently outperformed deeper ones. For instance, LL-3 typically exhibits better throughput than LL-5 or LL-7, reinforcing the notion that minimizing linked list depth is critical for performance.

The observed performance differences between the variants can be further explained by their cache behavior. In general, the LL variants generated a greater total number of cache references and cache misses, particularly at higher load factors. This behavior is largely due to bulkier write operations and more fragmented memory access patterns inherent to the pointer-based structure of linked lists. In contrast, the DA variants consistently exhibited lower L1 cache data load misses across most workloads, reinforcing their advantage in terms of memory efficiency and predictable memory access patterns compared to their LL counterparts. Furthermore, last-level cache misses were significantly higher in LL variants, especially in write-intensive scenarios. This indicates that although LL variants may seem more lightweight on a per-access basis, their inherently random, pointer-based organization leads to inefficient utilization of the cache hierarchy under high memory pressure. Overall, these cache-level observations provide a clear mechanistic explanation for the superior performance of DA variants in terms of both speed and memory efficiency.

Energy consumption measurements indicate that the LL-3 and DA-3 variants consistently exhibited higher energy usage across all benchmark scenarios. In contrast, the DA-7 variant demonstrated the most energy-efficient behavior on average, achieving the best overall balance between throughput and resource utilization. These results suggest that DA-7 may serve as a particularly suitable default choice in resource-constrained environments—such as battery-powered systems with processor characteristics similar to those used in our experiments—especially when the workload characteristics or concurrency levels are not known in advance. The improved energy efficiency of DA-7 is likely due to its optimized memory layout and reduced cache misses, which collectively minimize wasted computational effort and unnecessary memory accesses.

Prior works on concurrent hash maps largely emphasize designs that improve memory locality, scalability, and energy efficiency under contention [5,7,8,13,25–29]. In particular, Maier et al. [8] showed how an array-oriented design can achieve high throughput via predictable and cache-friendly accesses, while Pereira et al. [13] highlighted the importance of evaluating the energy efficiency of data structures in their most fundamental operations, such as *inserting*, *deleting*, and *iterating* over elements. Our contribution complements these lines of research by isolating separate chaining within a concurrent hash map design and providing a systematic comparison between linked lists and dynamic arrays. The scenarios evaluated in this study cover a wide and diverse range of real-world workloads, and the results are consistent with prior works, showing a clear difference between using linked list or dynamic array strategies for separate chaining in hash maps. Across all experiments, dynamic arrays variants consistently maximized throughput, maintained stable energy consumption (with no observed energy usage peaks), and leveraged cache memory more effectively than their linked list counterparts. We believe this work can serve as a practical reference for system developers and users when selecting a collision resolution strategy, helping to align data structure choices with workload characteristics and platform constraints.

## 6. Conclusions and 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. Dy-

namic 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 [41], impact the performance of the dynamic arrays approach.

**Author Contributions:** Conceptualization, A.C., M.A. and R.R.; Methodology, A.C., M.A. and R.R.; Software, A.C., M.A. and R.R.; Writing—original draft, A.C., M.A. and R.R.; Writing—review & editing, A.C., M.A. and R.R. All authors have read and agreed to the published version of the manuscript.

**Funding:** 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).

**Data Availability Statement:** No new data were created or analyzed in this study.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Maurer, W.D.; Lewis, T.G. Hash Table Methods. *ACM Comput. Surv.* **1975**, *7*, 5–19. [CrossRef]
2. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Addison-Wesley Longman: Boston, MA, USA, 2006.
3. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*; MIT Press: Cambridge, MA, USA, 2009.
4. Lehman, T.J.; Carey, M.J. A Study of Index Structures for Main Memory Database Management Systems. In Proceedings of the International Conference on Very Large Data Bases, Kyoto, Japan, 25–28 August 1986; pp. 294–303.
5. Tenenbaum, A.M.; Langsam, Y.; Augenstein, M.J. *Data Structures Using C*; Prentice Hall: Upper Saddle River, NJ, USA, 1990.
6. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed.; Addison-Wesley Longman: Boston, MA, USA, 1998.
7. Herlihy, M.; Shavit, N. *The Art of Multiprocessor Programming, Revised Reprint*; Morgan Kaufmann: Burlington, MA, USA, 2012.
8. Maier, T.; Sanders, P.; Dementiev, R. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.* **2019**, *5*, 1–32. [CrossRef]
9. 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 Proceedings of the IEEE International Conference on High Performance Computing, Data, and Analytics, Bengaluru, India, 18–21 December 2022; pp. 75–84.
10. David, H.; Gorbatov, E.; Hanebutte, U.R.; Khanna, R.; Le, C. RAPL: Memory power estimation and capping. In Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design, Austin, TX, USA, 18–20 August 2010; pp. 189–194.
11. Khan, K.N.; Hirki, M.; Niemi, T.; Nurminen, J.K.; Ou, Z. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*, 1–26. [CrossRef]
12. perf: Linux Profiling with Performance Counters. 2024. Available online: https://perf.wiki.kernel.org (accessed on 1 June 2025).
13. Pereira, R.; Couto, M.; Cunha, J.; Melfe, G.; Saraiva, J.; Fernandes, J.P. Paint Your Programs Green: On the Energy Efficiency of Data Structures. In *Composability, Comprehensibility and Correctness of Working Software*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 53–76.
14. Michanan, J.; Dewri, R.; Rutherford, M.J. Predicting data structures for energy efficient computing. In Proceedings of the International Green and Sustainable Computing Conference, Las Vegas, NV, USA, 14–16 December 2015; pp. 1–8.
15. Muralidhar, R.; Borovica-Gajic, R.; Buyya, R. Energy Efficient Computing Systems: Architectures, Abstractions and Modeling to Techniques and Standards. *ACM Comput. Surv.* **2022**, *54*, 1–37. [CrossRef]
16. Demaine, E.D.; Lynch, J.; Mirano, G.J.; Tyagi, N. Energy-Efficient Algorithms. In Proceedings of the ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, 14–17 January 2016; pp. 321–332.
17. Fatourou, P.; Kallimanis, N.D.; Ropars, T. An Efficient Wait-free Resizable Hash Table. In Proceedings of the Symposium on Parallelism in Algorithms and Architectures, Vienna, Austria, 16–18 July 2018; pp. 111–120.
18. Wang, J.; Liu, D.; Fu, X.; Xiao, F.; Tian, C. DHash: Dynamic Hash Tables With Non-Blocking Regular Operations. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 3274–3290. [CrossRef]
19. Prokopec, A.; Bronson, N.G.; Bagwell, P.; Odersky, M. Concurrent Tries with Efficient Non-Blocking Snapshots. In Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, New Orleans, LA, USA, 25–29 February 2012.

20. Shalev, O.; Shavit, N. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* **2006**, *53*, 379–405. [CrossRef]
21. Malakhov, A. Per-bucket concurrent rehashing algorithms. *arXiv* **2015**, arXiv:1509.02235. [CrossRef]
22. Scouarnec, N.L. Cuckoo++ hash tables: High-performance hash tables for networking applications. In Proceedings of the Symposium on Architectures for Networking and Communications Systems, Ithaca, NY, USA, 23–24 July 2018; pp. 41–54.
23. Baumstark, A.; Pohl, C. Lock-free Data Structures for Data Stream Processing. *Datenbank-Spektrum* **2019**, *19*, 209–218. [CrossRef]
24. Barnat, J.; Ročkai, P.; Štill, V.; Weiser, J. Fast, Dynamically-Sized Concurrent Hash Table. In *Model Checking Software*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 49–65.
25. Flajolet, P.; Poblete, P.; Viola, A. On the analysis of linear probing hashing. *Algorithmica* **1998**, *22*, 490–515. [CrossRef]
26. Pagh, R.; Rodler, F.F. Cuckoo hashing. *J. Algorithms* **2004**, *51*, 122–144. [CrossRef]
27. Celis, P.; Larson, P.A.; Munro, J.I. Robin hood hashing. In Proceedings of the Annual Symposium on Foundations of Computer Science, Portland, OR, USA, 21–23 October 1985; pp. 281–288.
28. Askitis, N.; Zobel, J. Cache-Conscious collision resolution in string hash tables. In Proceedings of the International Conference on String Processing and Information Retrieval, Buenos Aires, Argentina, 2–4 November 2005; pp. 91–102.
29. Askitis, N.; Sinha, R. HAT-trie: A cache-conscious trie-based data structure for strings. In Proceedings of the Australasian Conference on Computer Science—Volume 62, Ballarat, VIC, Australia, 30 January–2 February 2007; pp. 97–105.
30. Farach-Colton, M.; Krapivin, A.; Kuszmaul, W. Optimal Bounds for Open Addressing Without Reordering. In Proceedings of the Symposium on Foundations of Computer Science, Chicago, IL, USA, 27–30 October 2024; pp. 594–605.
31. Xu, S.; Liu, D. Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study. *Int. J. Networked Distrib. Comput.* **2015**, *3*, 60–68. [CrossRef]
32. Triplett, J.; McKenney, P.E.; Walpole, J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In Proceedings of the USENIX Annual Technical Conference, Portland, OR, USA, 15–17 June 2011.
33. Evans, J. A scalable concurrent malloc (3) implementation for FreeBSD. In Proceedings of the BSDCan Conference, Ottawa, ON, Canada, 12–13 May 2006.
34. Moreno, P.; Areias, M.; Rocha, R. On the implementation of memory reclamation methods in a lock-free hash trie design. *J. Parallel Distrib. Comput.* **2021**, *155*, 1–13. [CrossRef]
35. Moreno, P.; Areias, M.; Rocha, R. Memory Reclamation Methods for Lock-Free Hash Tries. In Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2019), Campo Grande, Brazil, 15–18 October 2019; pp. 188–195.
36. Moreno, P.; Areias, M.; Rocha, R. A Compression-Based Design for Higher Throughput in a Lock-Free Hash Map. In Proceedings of the 26th International European Conference on Parallel and Distributed Computing (Euro-Par 2020), Warsaw, Poland, 24–28 August 2020; pp. 458–473.
37. Moreno, P.; Areias, M.; Rocha, R. On Exploring Safe Memory Reclamation Methods with a Simplified Lock-Free Hash Map Design. In Proceedings of the Euro-Par 2024: Parallel Processing Workshops, Madrid, Spain, 26–30 August 2024; pp. 302–306.
38. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
39. Wang, C.; Hu, J.; Yang, T.; Liang, Y.; Yang, M. SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Boston, MA, USA, 10–12 July 2023; pp. 479–495.
40. Hu, D.; Chen, Z.; Wu, J.; Sun, J.; Chen, H. Persistent memory hash indexes: An experimental evaluation. *VLDB Endow.* **2021**, *14*, 785–798. [CrossRef]
41. Ben-David, N.; Blelloch, G.E.; Wei, Y. Lock-free locks revisited. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, 2–6 April 2022; pp. 278–293.