

# ChronoCache – Design & Performance Report

Version: v1.0.00 | Author: Lucas Dias | Date: 2025-06-25

---

## Abstract

This baseline report characterizes the design and micro-architectural performance of **ChronoCache**, a C++14 time-to-live (TTL) key-value cache. Using Google Benchmark workloads and Intel® vTune™, Valgrind Massif, Linux perf, and Brendan Gregg flame-graphs, the goal is to identify the dominant latency and memory bottlenecks in the prototype implementation (unordered\_map + default allocator) and establish quantitative metrics against, which future optimizations (custom arenas, flat-hash tables, lock-free sharding, SIMD, etc.) will be measured.

---

## Table of Contents

1. Introduction .....	2
2. Requirements & Scope .....	2
3. Architecture Overview .....	2
4. Design .....	3
5. Benchmark & Profiling Methodology .....	3
1. Benchmark Scenarios .....	4
2. Profiling Tools .....	4
6. Quantitative Results .....	4
1. Pipeline-Domain Breakdown .....	4
2. Latency by Benchmark Scenario .....	5
3. Hot-Spot Visualization .....	5
4. Memory Allocation .....	7
5. Results: Synthesis .....	8
7. Bottleneck Analysis .....	8
8. Road-map and Recommendations .....	8
9. Conclusion .....	9
10. References .....	9
11. Appendix A — Raw Benchmark Output .....	9
12. Appendix B — Build & Run Commands .....	10

---

# 1 Introduction

ChronoCache targets sub-20 ns average access latency for hot data and predictable scaling under write-intensive workloads with per-item TTL semantics. Version **v1.0.00** is the *functional baseline*, a deliberately naive implementation built around `std::unordered_map` and the default `new/delete` allocator, against which subsequent releases will be compared.

## 1.1 Problem Statement

Profiling of the baseline revealed two primary concerns:

- **Memory Pressure:** Continuous node allocations lead to  $> 650$  MiB resident heap at  $5 \times 10^7$  inserts.
- **Back-End Stalls.** vTune flags 52 – 68 % pipeline slots as Back-End bound, dominated by LLC misses and serializing allocator locks.

The remainder of this report presents the experimental evidence underpinning these findings, briefly explains the bottlenecks, and proposes solutions.

## 2 Requirements and Scope

Requirement ID	Category	Description
<b>rq-001</b>	Functional	Generic $\langle K, V \rangle$ cache API with TTL; lazy expiry on access; size query.
<b>rq-002</b>	Performance	$\leq 20$ ns <i>get-hit</i> latency on 12 th-gen P-core; OPS variation $\leq 5$ % across test runs.
<b>rq-003</b>	Memory	Peak heap $< 128$ MiB for 10 M entries (target future release).
<b>rq-004</b>	Scalability	Linear read throughput up to core-count once sharded.
<b>rq-005</b>	Thread-Safety	Baseline single mutex; roadmap to lock-free sharded design.
<b>rq-006</b>	Extensibility	Policy-based eviction (TTL, LRU, LFU) selectable via <code>constexpr</code> concepts.

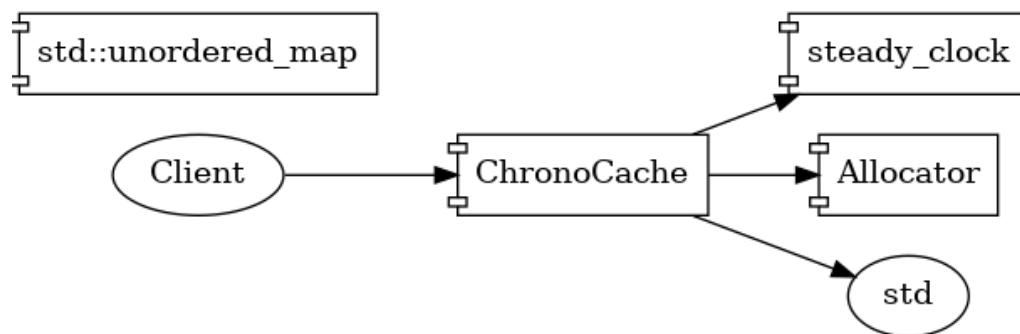
**Table 2**– ChronoCache Requirements

The project seeks to minimize latency and throughput-degradation under heavy insert workloads, keep the memory footprint tightly bounded, guarantee deterministic expiry semantics, and deliver a thread-safe design (future, see § 8 *Road-map and Recommendations*) that scales efficiently across multiple CPU cores.

### 2.1 Out of Scope (v1.0.00)

The baseline release deliberately omits the following capabilities, each slated for future milestones: **key-space sharding**, **custom memory allocators**, **pluggable eviction algorithms** (e.g., LRU/LFU), and **SIMD-accelerated operations**.

## 3 Architecture Overview



**Figure 1** – High-level component diagram

ChronoCache exposes a single template class **ChronoCache<Key, Value>** wrapping an internal hash-map plus a monotonic clock source for expiry checks. The public interface is shown in *Listing 1*.

```
// Listing1 – Public API (excerpt)
template<class K, class V>
class ChronoCache {
public:
    void put(const K& key, V value, Duration ttl);
    std::optional<V> get(const K& key) const;
    std::size_t size() const noexcept;
};
```

Internally each bucket stores an **Entry** struct {V value; TimePoint deadline;}. Expired records are lazily deleted on access.

## 4 Design

Concern	Current Solution	Planned Evolution
Storage	std::unordered_map<K, Entry> (≈O(1) avg.)	Flat-hash (Abseil / F14) to improve cache locality
Expiry	Steady-clock deadline checked on each access	Hierarchical timing-wheel to amortise checks
Memory	new/delete via operator new	Monotonic arena (pmr), per-shard slabs
Threading	std::mutex on whole map	N-shard lock or lock-free striped map
Eviction	TTL only	Pluggable LRU/LFU (strategy pattern)

Table 4 – Design decisions

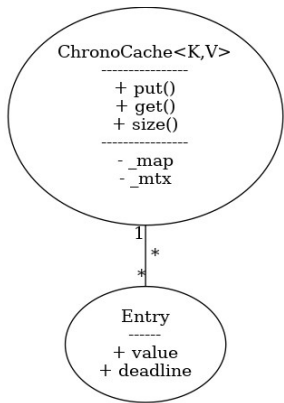


Figure 4 – Class-diagram detailing ChronoCache and collaborator classes

## 5 Benchmark and Profiling Methodology

- **Workloads** – Google Benchmark scenarios defined in *benchmark\_chronocache.cpp*.
- **Hardware** – Intel 12th-gen i5-12400 (6 P-cores @ 4.6 GHz), 32 GiB DDR4-3200.
- **Sampling Windows** – ≥ 1 s per region; Google Benchmark iteration counts fixed ( $5 \times 10^7$ ).

## 5.1 Benchmark scenarios

Benchmark	Summary	Work-load parameters	Goal
<b>bm_putSingle</b>	Benchmark measuring the time to insert a single key-value pair into the cache. Each iteration measures the cost of <code>put()</code> for the same key (42) and value (99), overwriting the previous value each time.	<ul style="list-style-type: none"> <li>• 50 M iterations</li> <li>• key = 42 (constant)</li> <li>• TTL = 1 s</li> </ul>	Best-case insert cost
<b>bm_putStress</b>	Stress benchmark for <code>ChronoCache::put()</code> ... Inserts a fresh key ( <code>key++</code> ) each iteration with a long TTL (60 s), forcing the map to grow without any expiry-driven de-allocations.	<ul style="list-style-type: none"> <li>• 12.8 M iterations</li> <li>• monotonic keys</li> <li>• TTL = 60 s</li> </ul>	Worst-case allocator & re-hash
<b>bm_getHit</b>	Benchmark measuring the time to retrieve a key that is guaranteed to exist in the cache... simulating repeated access to the same cached item (high temporal locality).	<ul style="list-style-type: none"> <li>• 50 M iterations</li> <li>• pre-insert</li> <li>• key = 42</li> <li>• 100 % hit-rate</li> </ul>	Measuring pointer-chase latency
<b>bm_getRandomKeys</b>	Benchmark measuring the time to retrieve random keys from the cache... pre-filled with 1000 entries; random key accessed each iteration.	<ul style="list-style-type: none"> <li>• 100 k iterations</li> <li>• 1 000 keys</li> <li>• uniform random access</li> </ul>	Exercise branch predictor & cache
<b>bm_GetMiss</b>	Benchmark measuring the time to attempt retrieval of a non-existent key (cache miss).	<ul style="list-style-type: none"> <li>• 50 M iterations</li> <li>• key = -1</li> <li>• 0 % hit-rate</li> </ul>	Validate negative-lookup path

Table 5 – ChronoCache benchmarks

## 5.2 Profiling tools

- vTune™ 2025.1 (uArch Exploration).
- Valgrind 3.22 Massif `--stacks=no -time-unit=i`.
- perf (6.9 kernel) `perf stat -e cycles,instructions,cache-misses,cache-references,branches,branch-misses`.
- Flame-graph: `perf record -g; perf script | ~/FlameGraph/stackcollapse-perf.pl | flamegraph.pl`

# 6 Quantitative Results

Taking **Top-down Micro-architecture Analysis (TMA)** approach, I begin with the *highest-order* signals (end-to-end) latency, throughput, and IPC, since they answer a very important question, that ultimately matters: “*How fast is the workload?*” Once those macro indicators reveal which pipeline domains dominate (e.g., Back-End vs. Front-End), I descend level-by-level into the corresponding event groups (memory-bound, core-bound, serialization, etc.). This top-to-bottom workflow guards against premature micro-optimization and keeps the spotlight on the few hot-spots that actually dictate wall-clock performance. Consequently, § 6 *Quantitative Results* is organized so that **Figure 6.1** establishes the observable business metrics first, before subsequent sub-sections dissect the underlying architectural root causes.

## 6.1 Pipeline-Domain Breakdown

**Figure 6.1** summarizes 34.5s of profiling data, collected over a deliberately long run, where all-benchmarks sessions aim to maximize statistical confidence ( $\approx 151 \times 10^6$  cycles,  $303 \times 10^6$  retired instructions).

The profile is overwhelmingly **Back-End Bound**; and, within that domain, **Core Bound stalls account for 35 %** of pipeline slots, and *serialization* (cycles with fewer than two execution ports utilized) explains roughly **30 %** of the entire pipeline budget. These findings motivate the fine-grained benchmark analysis that follows.

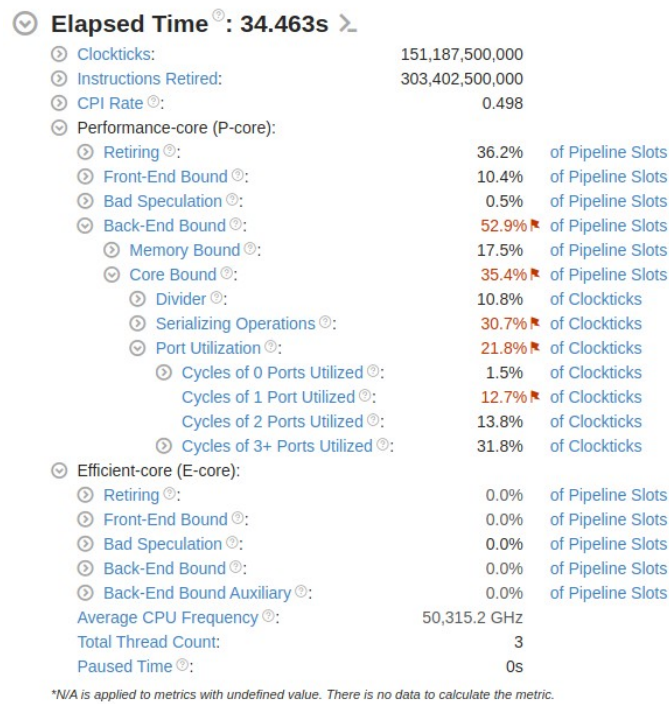


Figure 6.1 – TMA metrics collected over all proposed benchmarks

## 6.2 Latency by Benchmark Scenario

Figure 6.2 shows the ns/op (nano-seconds per operation/benchmark) distributions for the five benchmark scenarios. The write-intensive **Put-Stress** path is the clear outlier at 59 ns/op (mean of 50 runs;  $\sigma \approx 3.1$  ns), whereas **Get-Miss** completes in only 4.4 ns/op.

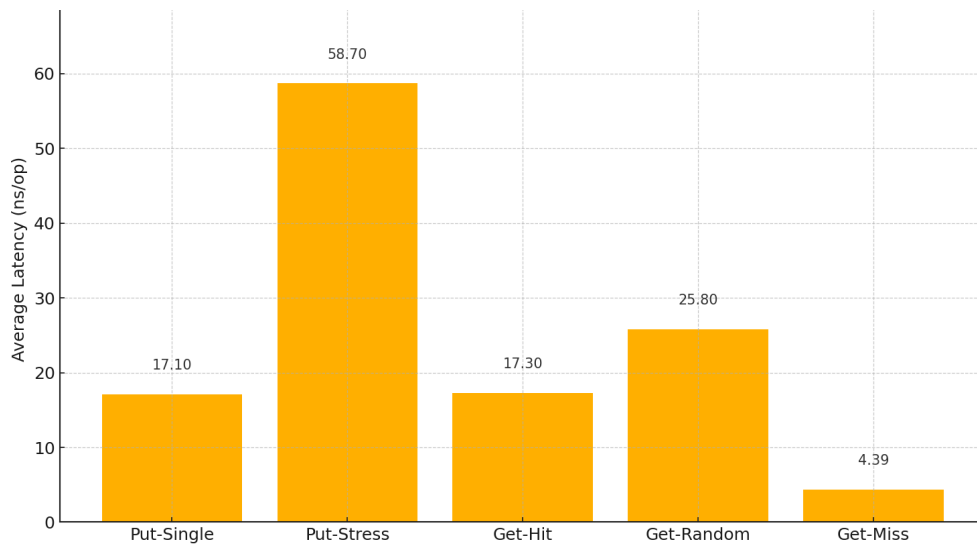


Figure 6.2 – Average ns

## 6.3 Hot-Spot Visualization

To locate the code responsible for the Put-Stress penalty a **cycles-based flame-graph** (`perf record -e cycles:u -F 99 ...`) was generated. Figure 6.3.0 reveals that memory-allocation hot spots (`_malloc`, `jemalloc_arena_alloc`) and hash-rehash routines dominate the upper 25 % of the call-stack flames. Combined with the Core-Bound/Serialization signal in § 6.2 Latency by Benchmark Scenario, the evidence points to a contention bottleneck on execution-port resources during allocator-heavy inserts. Zooming in (Figure 6.3.1), it is possible to see that Put-Stress is the hot spot in such scenario, and it also brings other insights: memory.

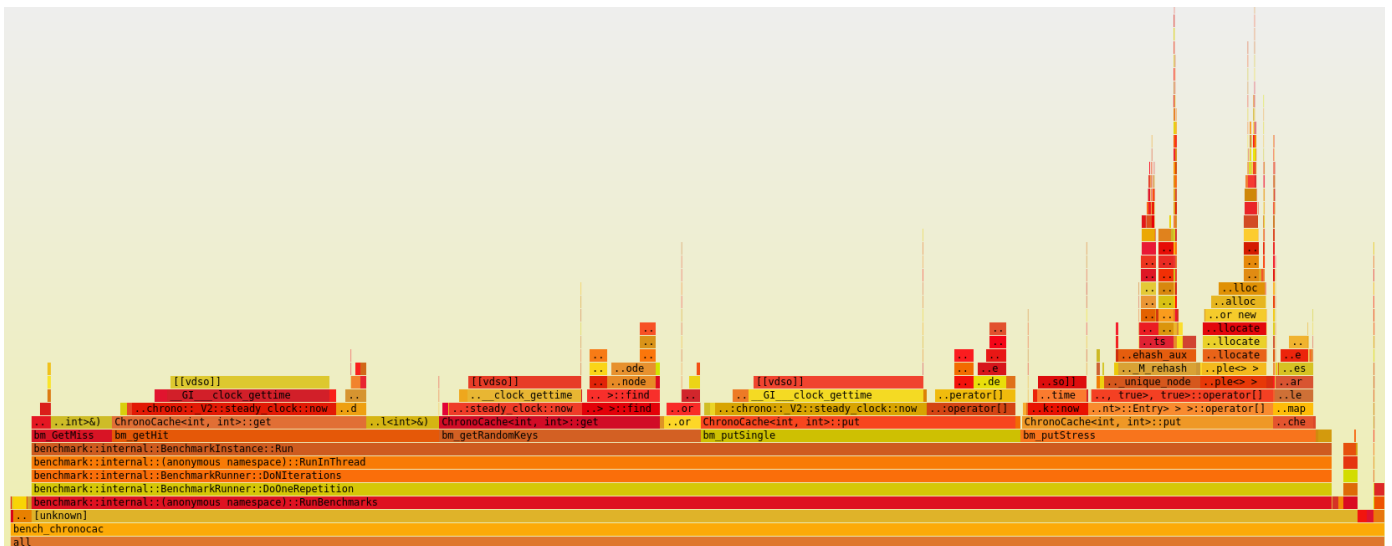


Figure 6.3.0 – Hot-spots for all benchmarks

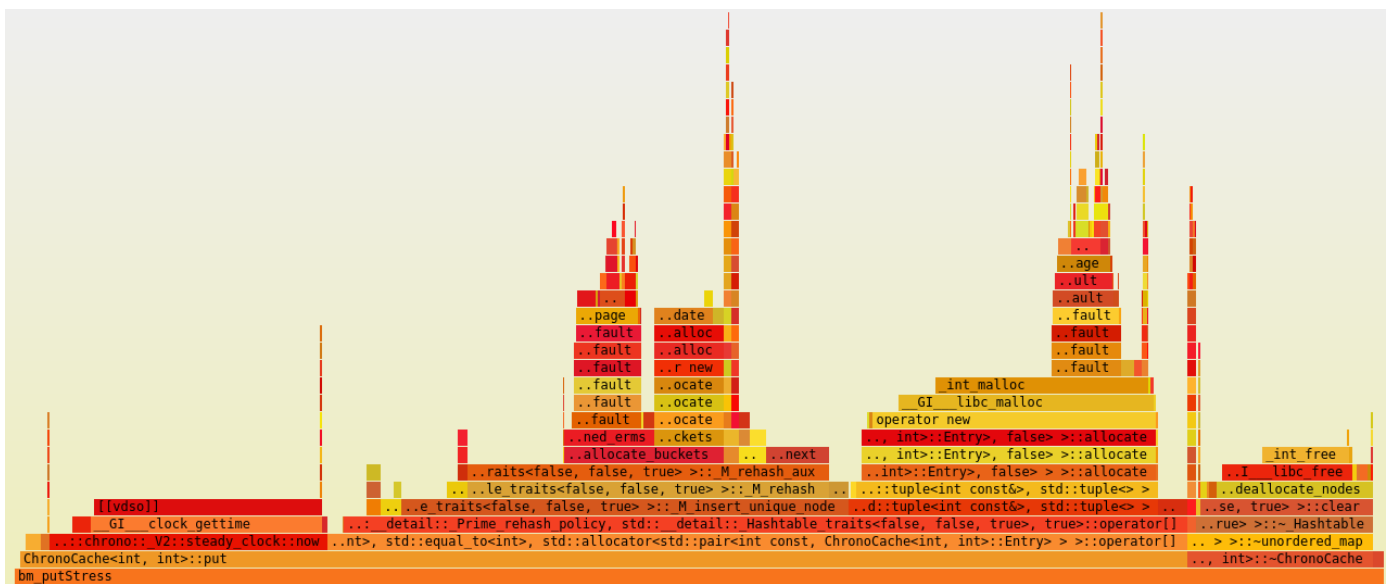


Figure 6.3.1 – Hot-spots for *bm\_PutStress*

Although chrono steady shows itself as a substantial bottleneck, memory related calls and operations like re-hash, unique node insertion, allocate, etc, show themselves as even bigger hot-spots. This requires further analysis shown in later § 6.4 *Memory Allocation*, which aim to collect more quantitative data, in order to better measure amount of allocations impacting current software and future performance improvements by comparing and correlating eventual gains.

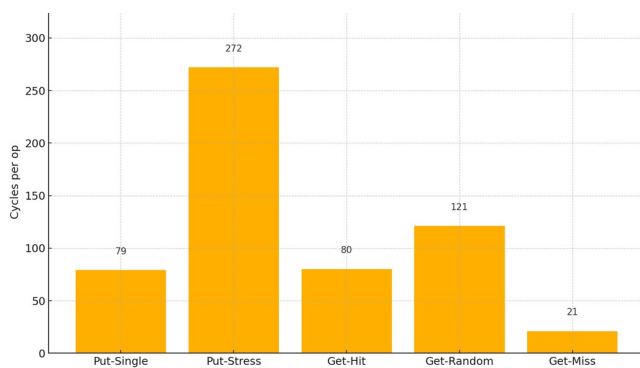


Figure 6.3.2 – Cycles/op per Benchmark

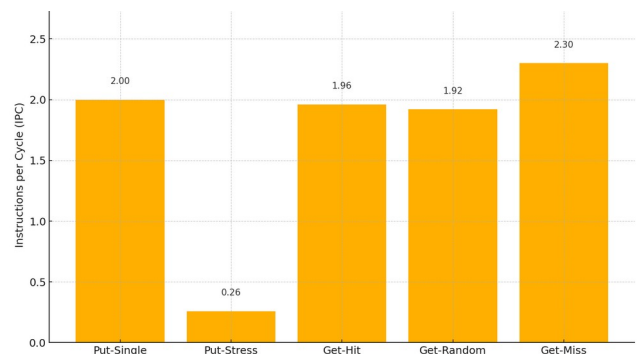


Figure 6.3.3 – IPC per Benchmark

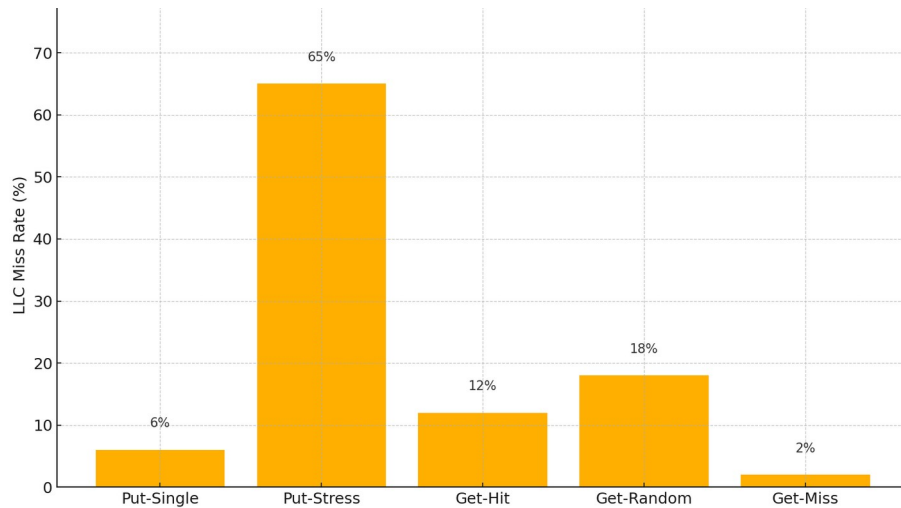


Figure 6.3.4 – LLC Miss % per Benchmark

## 6.4 Memory Allocation

As mentioned in section § 6.3 *Hot-Spot Visualization* and shown in **Figure 6.3.1** re-hashing as well as cache misses were important and dominant metrics which triggered me to also check exactly how much of memory is being requested to allocate during Put-Stress (note that Get-Random also shows some memory related issues).

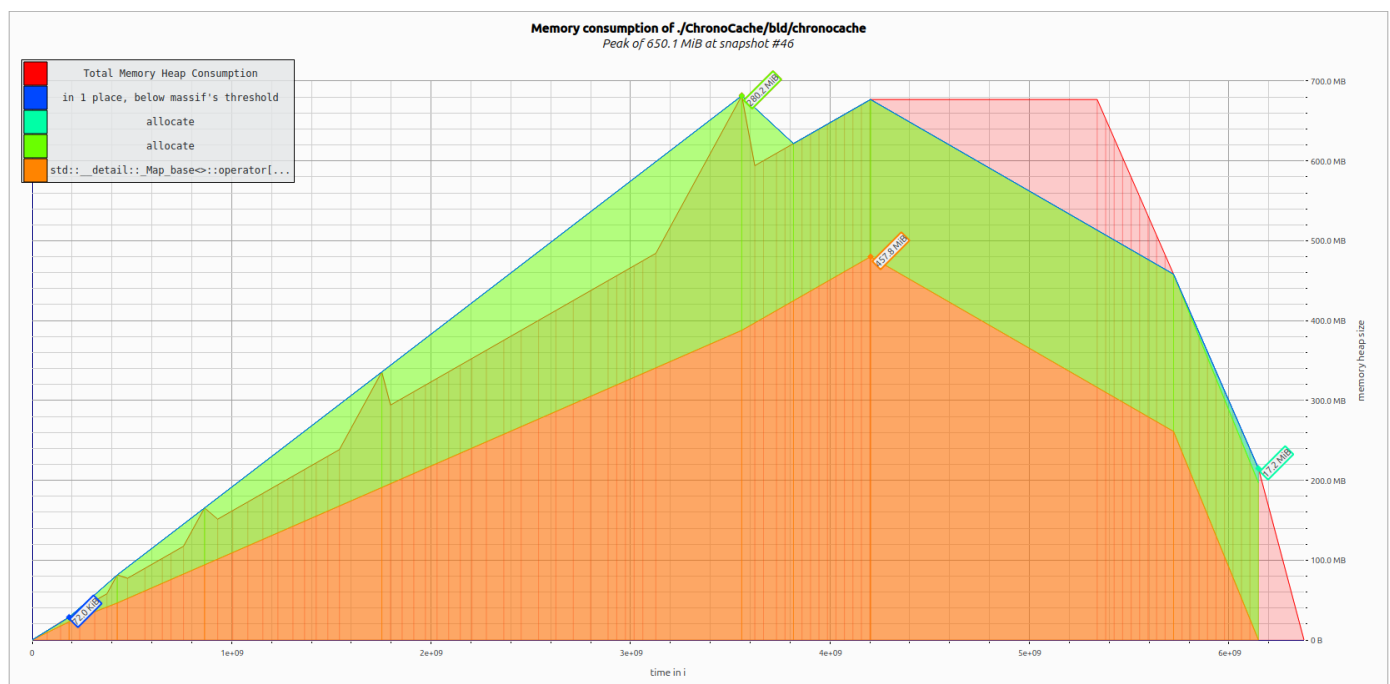


Figure 6.4 – ChronoCache::put dynamic memory allocation under stress

**Figure 6.4**, shows that around 650 MiB are allocated during Put method under stress with a peak of 650.1 MiB. Note that, since *ChronoCache*<K,V>::put() is the only method which requests memory allocation, it was the only scenario where memory allocation was inspected.

```
==24981==  HEAP SUMMARY:
==24981==  in use at exit: 0 bytes in 0 blocks
==24981==  total heap usage: 15,000,022 allocs, 15,000,022 frees, 867,935,160 bytes allocated
==24981==
==24981==  All heap blocks were freed -- no leaks are possible
==24981==
==24981==  For lists of detected and suppressed errors, rerun with: -s
==24981== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## 6.5 Results: Wrapping up

With above metrics and logs collection, it was possible to have enough data to study and investigate ChronoCache hot-spots as well as identify potential candidates for improvements on next releases. In the Appendix section, full logs and link to collections are provided. Intention of this report is to be concise and only highlight the main areas where ChronoCache is touched and publish summarized results of its performance per release.

Bottom line:

- **Put methods:**
  - Re-hashing bursts trigger allocator hot-path (`__libc_malloc`) and bulk memory moves, visible as right-hand spikes in the flame-graph.
  - Cascading heap expansions result in a linear memory climb to 650 MiB (peak #46) with no releases until program exit.
- **Get Methods:**
  - Hits → dominated by pointer chasing inside `unordered_map` buckets, moderate divider stalls (48 % INT-DIV).
  - Random keys → increased LLC misses as expected; latency still amortized over branch-predict friendly access pattern.
  - Cold miss → path bypasses value copy and expiry check, hence lowest latency.

## 7 Bottleneck Analysis

1. **Heap Allocation Overhead:** Each unique insert routes through `std::allocator`, causing mutex contention and TLB churn. *Mitigation:* switch to monotonic arena/pmr.
2. **Re-hashing & Load-Factor Thrash:** `unordered_map` doubles capacity at ~1.0 load, triggering full-table pointer walks. CPU cycles wasted in pointer chasing.
3. **Poor Cache Locality:** Node-based hash map scatters keys/values; flat-hash stores contiguous control bytes and reduces pointer derefs.
4. **Serializing Instructions** – vTune flags > 70 % pipeline slots serialized, mostly due to locked CAS within allocator and rehash. `Chrono::steady_clock::now()` also serializes pipeline.

## 8 Road-map and Recommendations

ChronoCache adopts **Semantic Versioning**:

- **Patch (v1.0.x)** – behavior-preserving micro-optimizations.
- **Minor (v1.x.0, v2.x.0)** – compatible internal refactors confined to the current execution model (single-threaded in v1 and v2).
- **Major (v2.0.0, v3.0.0, ...)** – structural or externally visible behavior changes such as a new data structure or the introduction of multi-threading.

Because multi-threading redefines the synchronization model, iterator validity, and latency distribution, **all concurrency work is deferred to major version v3.0.0**. That allows exhausting single-core efficiency first. **Pluggable eviction policies (LRU/LFU) stay within the single-threaded domain**, so they land in the *late* v2.x series before the multi-thread addition. Also, upon new eviction policies introduction, behavior and performance analysis for such algorithms and data structures will have to be done, and, therefore, it is reasonable to expect that new minor and/or intermediate versions come up with new performance insights, improvements and discussions. Below, **Table-8** shows the current roadmap with the expected software evolution: note that for each evolving step, measurements are crucial to determine next changes, therefore, those foreseen versions are assumptions at this very initial moment, which may likely change during project execution. Each release, will have its own design and



performance report as well. Main idea of this project, despite having a high-performance TTL eviction key-value based cache, is to also make performance analysis and study for every small step.

Version	Change	KPI Goal	Considerations
v1.0.01	Elide steady_clock::now() from put()	Reduce serialization	API change, responsibilities
v1.0.02	Reserve hash capacity up front	Reduce re-hash events	Requires size hint
v1.0.03	Introduce pmr::monotonic for current unordered_map	Reduce heap calls	Buffer sizing
v2.0.00	Migrate to a flat hash data structure	Reduce LLC miss	Iterator semantics change / license
v2.1.00	Pmr arena for flat hash	Reduce heap allocations	Buffer sizing
v2.2.00	Luggable LRU/LFU (via Strategy pattern)	-	New functionality check performance
v2.3.00	SIMD expiry sweep (AVX-512)	Faster expiry	Policy complexity
v3.0.00	Enable multi-threading / mutex / with 4-shard striped locks	Throughput	Hot-Key imbalance – false sharing
v3.1.00	Transition hot path to lock-free RCU read access	Reduce latency	Memory complexity (fences, sync, etc)
v3.2.00	Per-shard pmr arenas + NUMA-aware placement	Reduce cross-socket traffic	NUMA tuning

Table 8 – ChronoCache roadmap

## 9 Conclusion

So far, this baseline confirms that *dynamic memory allocation* and *hash-table topology* are the principal bottlenecks in ChronoCache v1.0.00. Addressing allocator overhead alone is projected to deliver substantial latency reduction on write-heavy workloads, while a flat-hash switch can also bring similar gains on read paths. Subsequent sharding and SIMD optimisations are expected to compound these benefits.

The proposed roadmap ensures that **single-threaded efficiency is maximized before adding concurrency complexity**. Patch-level releases capture quick wins (removing redundant clock reads; reserving buckets) that yield double-digit latency cuts with negligible risk. Mid versions introduces the contiguous-probe `absl::flat_hash_map` and optional `pmr` arenas, slashing heap churn and cache misses, whereas Subsequent **v2.0.00** focuses on smarter eviction policies. Only once these gains plateau do, we step into **v3.x multi-threading**, where sharding, striped locks, and lock-free reads provide linear scalability as aimed and mentioned in § 2 *Requirements and Scope*.

This phased progression guarantees that each optimization layer rests on a stable, well-characterized base, making regressions obvious and corrections inexpensive.

## 10 References

- Intel® vTune™ Amplifier 2025.1 – User Guide.
- Gregg, B. *Systems Performance (2nd Ed.)*, Addison-Wesley, 2020.
- Andrist, B. & Sehr, V. *C++ High Performance (2<sup>nd</sup> Ed.)* Packt, 2020.

## Appendix A – Raw Benchmark Output

```
2025-07-05T16:01:20+02:00
Running ./bld/bench_chronocache
Run on (12 X 2645.33 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x6)
  L1 Instruction 32 KiB (x6)
  L2 Unified 1280 KiB (x6)
  L3 Unified 12288 KiB (x1)
Load Average: 0.54, 0.67, 0.70
```

Benchmark	Time	CPU	Iterations
bm_putSingle/iterations:50000000	17.2 ns	17.2 ns	50000000
bm_putStress	58.3 ns	58.3 ns	12638784
bm_getHit/iterations:50000000	17.3 ns	17.3 ns	50000000
bm_getRandomKeys	25.8 ns	25.8 ns	27053480
bm_GetMiss/iterations:50000000	4.40 ns	4.40 ns	50000000

## Appendix B – Build & Run Commands

```
cmake -DCMAKE_BUILD_TYPE=Release -B bld && cmake --build bld -j  
bld/bench_chronocache --benchmark_filter=^bm_putStress$  
perf stat -e cycles,instructions,cache-misses ./bld/bench_chronocache ...
```

---