

Microarchitectural Characterization of High-Capacity Hash Tables on Modern Server Hardware

Peter Oslington and Shoaib Akram

Abstract—Hash tables are essential to modern online services, providing constant-time access to critical user data. Modern hash tables are highly optimized. For example, the Swiss hash table from Google performs several lookup optimizations to speed up the front end of their services. Unfortunately, in prior literature, there is little understanding of how these optimizations interact with modern processors with sophisticated hardware speculation and cache and memory hierarchy. This work uses a rigorous (top-down) performance analysis methodology to uncover key bottlenecks in accessing different implementations of high-performance hash tables. We find that: (1) on find operations for elements that exist in the table, backend restricted stalls such as memory and core bound stores are the key bottlenecks with over 60% of benchmark time spend stalled due to backend capacity on benchmarks not effectively using the cache, (2) on attempting to find elements not in the table substantial stalls are present due to both backend bound stalls and branch mispredictions with 15% - 57% of stalls being due to backend restrictions and 12% to 46% of stalls being due to branch restrictions, (3) for hash tables that have sizes larger than last level cache (Level 3 Cache), the backend bound stalls are primarily due to main memory bottlenecks, but tables that fit within the Level 3 cache are primarily bound by the level 3 cache, and (4) hash table access patterns where a small set of elements are accessed repeatedly are significantly faster due to 65% - 90% of CPU execution slots being effectively used. We validate our finding across several key parameters, including table element size, high and low table density, and access patterns. Our work opens up state-of-the-art hash table implementations to further optimizations informed by our analysis.



1 INTRODUCTION

Hash tables are used across a wide range of modern computing workloads, such as databases [1], search engines and key-value stores [2] due to their fast (generally constant time) operations. These hash table-based components of software can for substantial portions of large-scale compute operations, such as at Google, where in 2017, 1% of CPU time and 3% of memory was used by hash tables [3].

As such, small optimisations to all hash table operations can have important impacts on large-scale computing, but there has been limited detailed benchmarking of hash table workloads analysing key bottlenecks to hash tables. Many attempts have been made to design more efficient hash table algorithms by both industry [3] and academic research [4]–[6] through strategies such as algorithmic optimisations, adapting of tables to new technologies such as persistent memory [7]–[9], as well as making micro-architectural changes to optimise hash tables [1], [10], but many of these have been lacking a low-level analysis of the bottlenecks attempting to be solved, particularly across a large range of possible hash table parameters.

In this paper, we conduct large-scale benchmarks of high-performance C++ hash tables and provide micro-architectural explanations of key performance differences between hash table implementations across a range of parameters including value size, table size and threading behaviour. Applying Intel’s Top Down performance analysis, which makes use of Intel CPU performance counters to provide a detailed breakdown of microarchitectural behaviour demonstrates that hash table workloads are primarily backend bound, due to limitations of external memory and last-level cache limitations cache capacity limitations. However, in some benchmarks such as finding elements not contained in the table, frontend restrictions and branch misspeculation can substantively impact the performance of hash table accesses.

The benchmarks conducted primarily target the performance behaviour on hash table accesses, so benchmarks operations such as finding elements in the table, finding elements not in the table, and inserting elements already contained in the table. The benchmarks were carried out on three high-performance hash table implementations, `absl::flat_hash_set`, `absl::node_hash_set` and `google::dense_hash_set`. The tables were benchmarked across a wide range of parameters with tables ranging up to 64GiB of total capacity, across a range of value sizes, access patterns and table density.

The key findings of this work are:

- On benchmarks where a single element is accessed repeatedly (hot caching), the benchmark utilises the available microarchitectural resources effectively, with between 60% - 90% of available execution going to usefully retired instructions. However, on benchmarks where accesses iterate across every element in the table (cold caching), both the performance and the fraction of successfully retiring instructions are significantly reduced, with at most 10% - 15% of instructions retiring.
- On find operations for elements that exist in the table, backend restricted stalls such as memory and core bound stores are the key bottlenecks. On cold caching mode, backend bound forms 60% - 85% of the performance bottlenecks.
- On find operations targeting elements not in the table, cycles wasted due to branch mispredictions become substantial, accounting for at least 10% and up to 45% in some benchmarks.
- For hash tables where the total capacity can fit the last level CPU cache, the backend bound performance bottleneck is primarily due to last level cache bottlenecks, but larger hash tables become heavily bound by external memory (RAM).

• P. Oslington and S. Akram are with the Australian National University, Canberra. E-mail: peter.oslington@anu.edu.au, shoaib.akram@anu.edu.au

2 RELATED WORK

Considerable work exists attempting to optimise hash table algorithms due to their wide range of uses. This includes attempts to improve the algorithm such as robin hood hashing [5], hopscotch hashing [4], and cuckoo hashing [6]. However, most attempts at optimisation have lacked a detailed analysis of performance bottlenecks that make it difficult to know how effectively measure how effectively these changes have resolved bottlenecks. As such, this research attempts to fill the gap in existing research by providing detailed, wide-ranging performance analysis.

This is particularly relevant for microarchitectural optimisations [1], [10], where there is a lack of detailed performance analysis across a wide range of parameters. While some research [1] does provide microarchitectural analysis, it is lacking detailed methodology and discussion of microarchitectural results, as well as targeting a limited set of hash table parameters. We also find substantial discrepancies between the data measured here when compared to the microarchitectural breakdown provided by Zhang and Sanchez [1], which over-represents the branch misprediction component of hash table performance.

3 BACKGROUND

3.1 Topdown Analysis Methodology

In order to conduct an effective analysis of micro-architectural bottlenecks, Intel’s Top Down Performance Analysis Methodology [11] was used. The top-down methodology provides a workflow of breaking down each available pipeline slot in a modern Intel CPU into a category. The first level of top-down analysis breaks uops between four categories, frontend Bound, bad speculation, retiring and backend bound. The full breakdown provided by the top-down analysis methodology can be seen in figure 1.

The analysis in this paper provides this first level of top-down analysis across all benchmarks performed, with a breakdown of backend bound slots into cache bound and each level of cache bound for most parameters used.

As hash tables are a heavily data-based workload, it is likely that significant bottlenecks may be coming from the memory and therefore backend bound categories. However, there is also scope for bottlenecks from the front end or bad speculation, particularly in situations where more complex logic needs to happen like resolving a hash collision.

3.2 Potential Bottlenecks

The data presented in the paper primarily targets hash finding operations, such as finding elements or inserting elements already in the table. Generally hash tables provided constant time access to elements, as the hash provides a direct memory offset to find the element. However, the constant time accesses of hash tables begin to break down when a collision occurs - when two elements are inserted into the table which has the same hash value, and therefore should be stored at the same index in the hash table. These collisions are generally resolved in two main ways. A chaining hash table will create a linked list of colliding values, storing a pointer to the linked list in the hash table, whereas a probing hash table will find the next available slot in the table and store the element there [3]. All tables benchmarked here are probing hash tables, storing values directly in the hash tables, rather using pointers or linked lists [3].

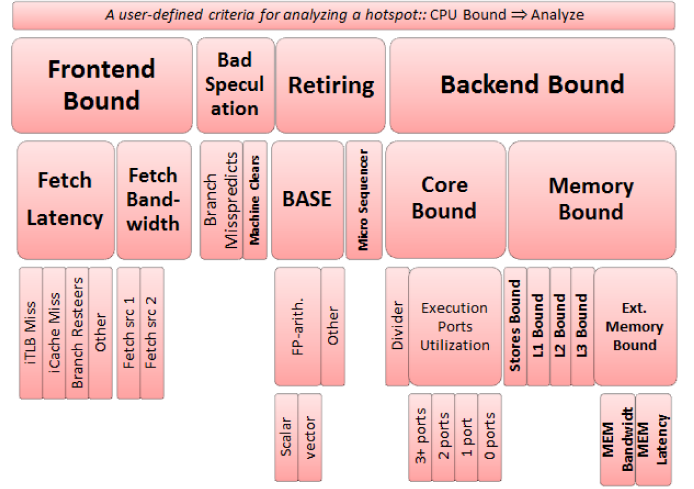


Fig. 1. The Top-Down Analysis Hierarchy [11]

Both of these approaches result in the need to loop through each element in a bucket of colliding values, which leads to two major categories of find operations, which may have distinct performance characteristics. When finding elements that are contained in the table, the find can exit as soon as it finds the matching element, whereas when finding an element not in the table requires checking every element in the bucket before the find can fail. Therefore, attempting to find elements not in the table is likely to result in, on average, much more loops and iterations than finding an element in the table. As the ordering of the buckets is generally based on the order in which elements were added and removed to the table, the point at which the looping over elements will terminate is challenging to predict, so is likely to lead to increased load on the frontend and branch predictor. It is also likely that backend restrictions will provide a substantial limitation to all hash table workloads, due to the heavy data and memory-based nature of hash tables.

4 HASH TABLES

Three modern, high performance hash tables were benchmarked as part of this analysis:

- `absl::flat_hash_set` [12]
- `absl::node_hash_set` [12]
- `google::dense_hash_set` [13]

These hash tables were selected based on hash tables that were in use or being rolled out at Google in 2017 [3], as well as wider consensus on effective high performance C++ hash table implementations. All three tables were originally developed by Google, and have since been open-sourced [3].

5 METHODOLOGY

5.1 Hardware

Benchmarks were carried out on the hardware specified in table 1.

5.2 Software

All benchmarks were carried out using a hash table benchmark tool developed by Google [14], which provides a framework for testing C++ hash tables over a range of parameters.

The three tables listed in section 4 were benchmarked across three key workloads. Find operations were tested both in cases

TABLE 1
Benchmark hardware specifications

System	
Operating System	Ubuntu 20.04 Linux
Processor	
Processors	Intel® Xeon® Gold 6300 (Ice Lake)
Architecture	3rd generation Xeon (Ice Lake)
Number of processors	2
Number of cores	28 physical cores (56 logical)
Core frequency	2.6GHz
Issue width	5-wide
Cache Hierarchy	
L1-I	48KiB per core
L1-D	32 KiB per core
L2	1.25 MiB per core
L3	42MiB, shared
DRAM	
Capacity	512GB
Data rate	2666 MT/s

where the element being targeted was included in the table (FindHit), and where the element was not present in the table (FindMiss). Insert operations were tested only in cases where the element is already in the table (InsertHit), to prevent the benchmarks from modifying the size of the table as it runs. These benchmarks provide a good representation of the retrieval behaviour in each of the hash table implementations used, but only provide a partial understanding of the insert behaviour of the tables, as element insertion never occurs in the benchmarks used. The benchmark also attempts to minimize the impact of the hashing function on benchmarks by using a consistent very low cost hash function across all benchmarks.

5.3 Initial wide-ranging benchmarking

Initial benchmarks were carried out across the following range of parameters:

- All three hash table implementations listed in section 4.
- All three benchmark operations, FindHit, FindMiss and InsertHit.
- Table sizes ranging from 2^4 to 2^{20} elements.
- Value sizes ranging from 2^2 to 2^{16} bytes.
- Cache hot or cold, where the same element is accessed repeatedly (hot), vs iterating over the whole table (cold).
- Maximum and minimum table density, where the table is about to resize larger (max), or has just resized (min).
- Each benchmark access repeated 20 000 000 per benchmark run
- Each benchmark run repeated 5 times, with results averaged across the 5 runs.

The benchmark tool used [14] uses Google’s general purpose C++ benchmark library [15], which provides support for Intel hardware performance counter through the libpfm library [16]. The performance counters and metrics used were primarily based of Intel’s existing publications on performance counters from 2014 [11]. However, due to the changes in architecture and performance counter support, a series of changes were made to the counters used to simplify and increase the accuracy of the metrics used. The full details of the performance counters used can be found in appendix A.

The following metrics were used to represent each of the major categories of the Top Down methodology:

TABLE 2
Level 1 top down metrics used [11]

Metric Name	Formula
Frontend Bound	FetchBubbles / TotalSlots
Bad Speculation	(SlotsIssued – SlotsRetired + RecoveryBubbles) / TotalSlots
Retiring	SlotsRetired / TotalSlots
BackendBound	1 – (Frontend Bound + Bad Speculation + Retiring)

5.4 More detailed top-down analysis

Level 3 of the topdown analysis methodology was also used across the range of parameters listed above targeting memory-based stalls, using the metrics found in table 3.

TABLE 3
Level 3 memory top-down metrics used [11]

Metric Name	Formula
L1 Bound	(MemStalls.AnyLoad - MemStalls.L1miss) / Clocks
L2 Bound	(MemStalls.L1miss - MemStalls.L2miss) / Clocks
L3 Bound	(MemStalls.L2miss - MemStalls.L3miss) / Clocks
Ext. Memory Bound	MemStalls.L3Miss / Clocks
Memory Bound	(MemStalls.AnyLoad + MemStalls.Stores) / Clocks

5.5 Threaded analysis

The benchmark was carried out again with a more limited set of parameters targeting the multithreaded behaviour of hash table accesses, with the parameters found in table 3.

- All three hash table implementations listed in section 4.
- All three benchmark operations, FindHit, FindMiss and InsertHit.
- A table size of 2^{16} elements.
- A value size of 2^{10} bytes.
- Hot and cold caching modes.
- Maximum and minimum table density.
- Numbers of threads ranging from 1 to 112 threads.

6 RESULTS

A comparison of the relative performance of the three hash tables benchmarked is presented in figure 2. A value size of 64 bytes was selected as a plausible typical value, in a similar range to the benchmarks carried out by Zhang and Sanchez [1], but of sufficient size to ensure behaviours could be measured both on tables smaller and larger than L3 cache capacity.

Level 1 Topdown Analysis is presented for a hash table containing 256-byte values in figures 3-5, across each hash table, cache mode and fill density. The absolute time for each benchmark is presented in black.

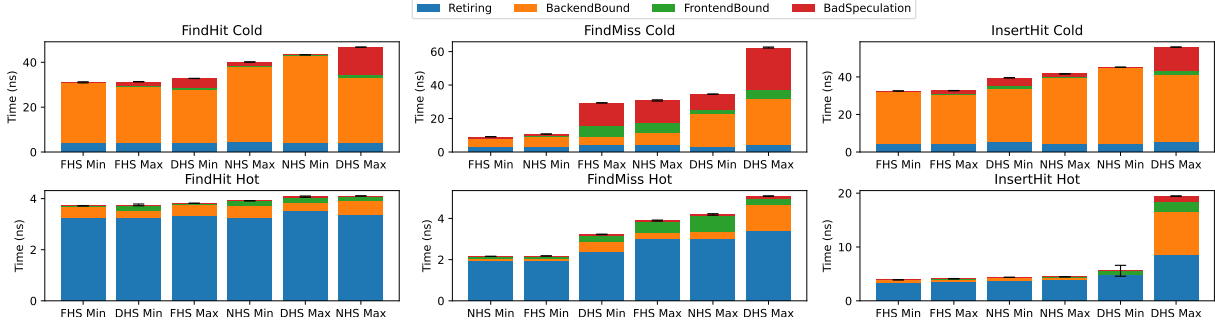


Fig. 2. Relative time and topdown breakdown for `absl::flat_hash_set` (FHS), `absl::node_hash_set` (NHS) and `google::dense_hash_set` (DHS), at maximum and minimum table density, with a value size of 256 bytes, and around 9×10^5 elements, for a total table size of approx. 55 MB. Standard deviating across the 5 repeated runs is presented as error bars on the char

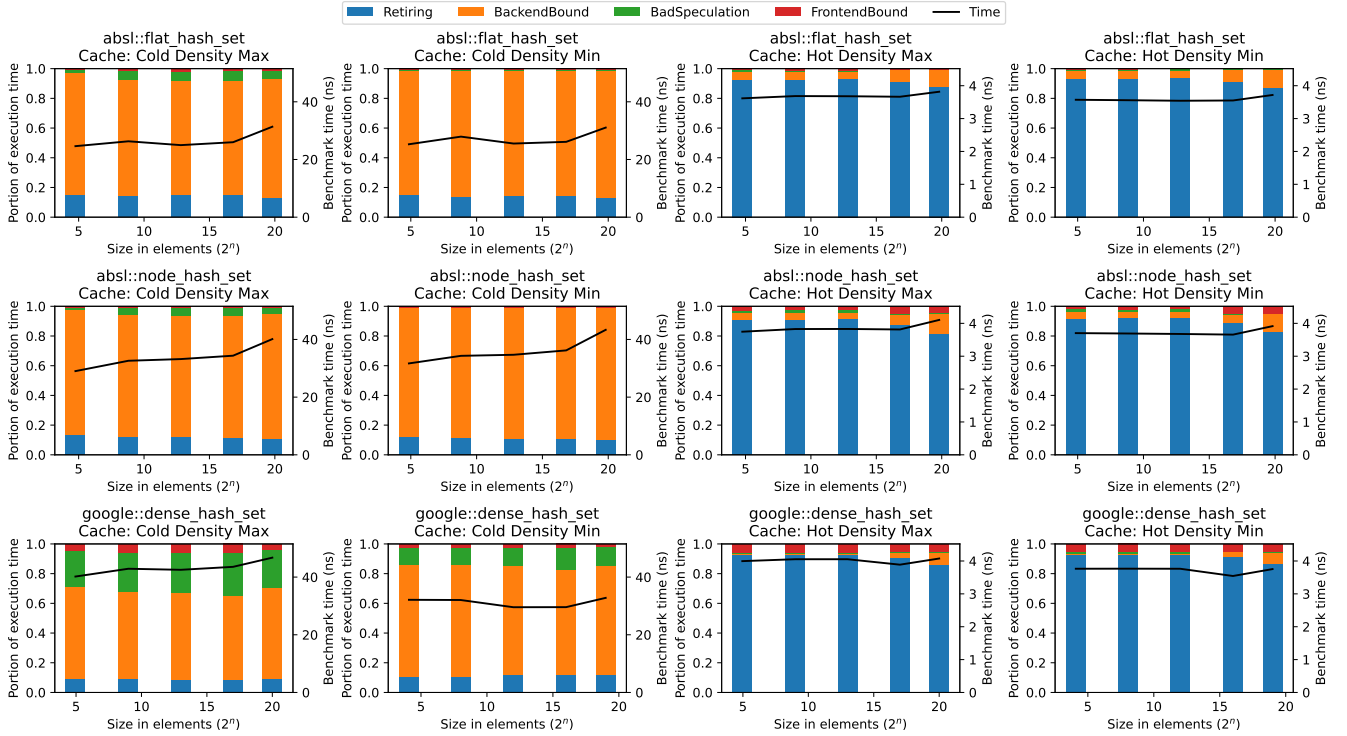


Fig. 3. Level 1 Top Down Performance analysis for 256-byte value hash tables on the FindHit Benchmark. The absolute time for each benchmark is presented in black.

6.1 Performance Analysis

In the benchmarks displayed in figure 2, `absl::flat_hash_set` at minimum fill density provides the strongest performance across almost all benchmarks tested, with `absl::node_hash_set` being slightly faster on finding elements not in the table in hot cache mode, and `google::dense_hash_set` generally performs the worst, with substantial performance losses on finding elements not in the table in cold cache mode.

Across all benchmarks we see that flat hash set in its best performance case has only memory-bound bottlenecks. As the overall performance decreases, there is an increase in time spent in backend-bound stalls, as well as the introduction of stalls due to the branch predictor and front end.

The significant performance difference between the low and high-performing access in figure 2 suggests that there is substantial scope for improvements to hash table operations through improvements to the memory.

6.2 Cache Mode

A key observation across all benchmarks is the clear distinction between hot and cold cache benchmarks - where the benchmark is either iterating over the same element of the hash table repeatedly compared to iterating across every element of the table. When repeatedly accessing the same element, very high levels of retiring instructions are achieved, with some benchmarks reaching 90% retiring uops, and all benchmarks achieving at least 60% uop retirement. This places repeated access to hash tables comparable to well-optimised high-performance computing applications [17]. Therefore, there is little room for further optimisation from a micro-architectural perspective for repeated hash table access. Further optimisations would need to come from optimisations to the algorithms being used, which is beyond the scope of this analysis.

Therefore, further single-threaded analysis will target the cold cache mode, as it has the most scope for optimisation due to the much lower successful retirement, generally $< 20\%$.

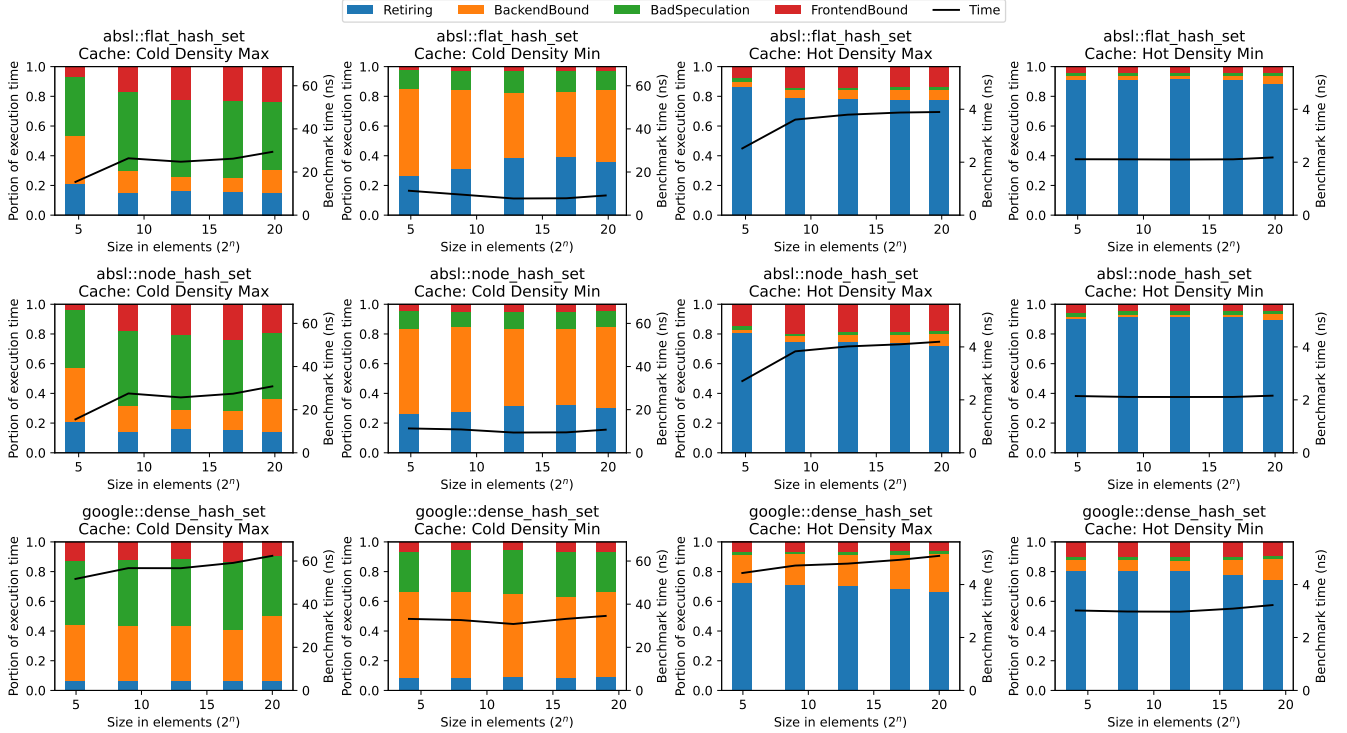


Fig. 4. Level 1 Top Down Performance analysis for 256-byte value hash tables on the FindMiss Benchmark. The absolute time for each benchmark is presented in black.

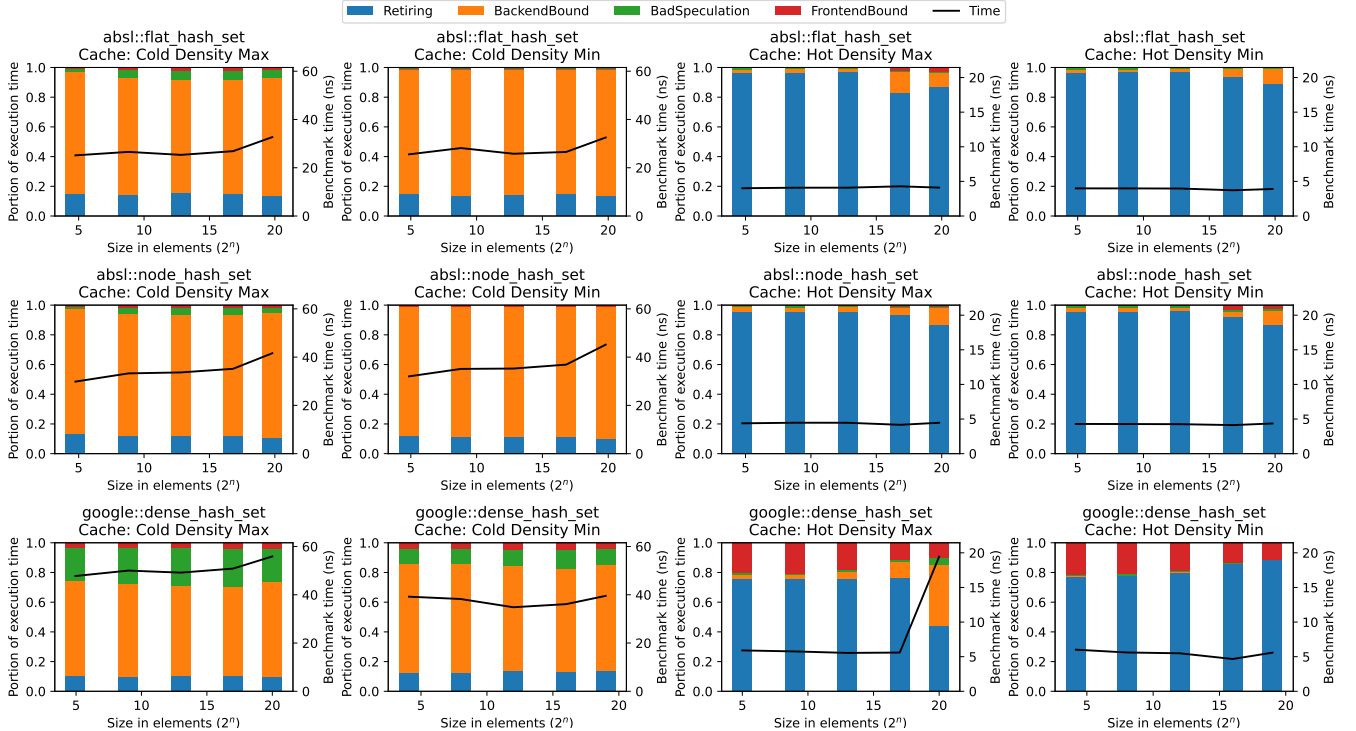


Fig. 5. Level 1 Top Down Performance analysis for 256-byte value hash tables on the InsertHit Benchmark. The absolute time for each benchmark is presented in black.

6.3 Backend Bound

A recurring trend across all cold cache benchmarks is the strong recurring presence of the backend-bound category. This category is present as a substantial fraction of wasted cycles but is particularly prominent in the FindHit and InsertHit benchmark,

where the backend bound category consistently accounts for $> 50\%$ of total available pipeline slots. Backend bound can be a result of either core restrictions, where instructions are unable to executions due to missing data from previous instructions such as a long chain of dependent arithmetic operations, or

due to memory bottlenecks [11].

6.4 Frontend & Speculation Bottlenecks

While the most consistent performance bottlenecks are due to backend-bound considerations, there are some contexts in which frontend limitations and bad branch predictions. This is most clearly present in the FindMiss benchmark but is present in the dense hash set performance across all benchmarks. The presence of bad speculation in find miss benchmarks is anticipated due to the unpredictable branching behaviour of looping over each element of the table. The presence of bad speculation in the top level analysis of the dense hash set indicates that the implementation is using less consistently predictable branch operations, even when the element is contained in the table.

6.5 Multi-threaded analysis

A representative extract of the multi-threaded data is presented in figure 6. Data for the InsertHit benchmark, minimum table density and the node hash set were not included due to similarity to included data. We can see that there is a substantial drop-off in hash table performance when running many simultaneous hash tables compared to a singular

The key observation from the threaded data presented in figure 6 is the substantial growth in the backend bound category as the threading increase, indicating the benchmark is becoming significantly more core or memory bound as it is run simultaneously across large numbers of threads. This is particularly evident in the hot cache benchmarks, where the rate of successful retirement drops off from $> 80\%$ on single-threaded tests to $< 10\%$ of execution time on large multithreaded tests. As the increase in threading will not increase the likelihood of dependant chains of instructions that would cause core-bound restrictions, it is likely that this additional wasted execution time is due to memory-bound restrictions. This may be due to L3 cache collisions in the shared L3 cache, or it could be due to either latency or bandwidth restrictions on external memory. Further threading-specific analysis would be needed to properly identify the source of these restrictions, but it seems likely that the drilled-down backend analysis carried out for single threaded tests would share some characteristics with the multithreaded benchmarks.

6.6 Memory Bound Drill Down

A subset of the data collected at the level 3 top-down analysis methodology is presented in figure 7. In the data presented, the source of memory-bound limitations is either L3 cache or memory-bound limitations. In the smaller tables, ($< 2^{12}$ elements for most table implementations), we see that the table is primarily L3 cache bound. With 16KiB elements and 2^{12} elements, the resultant hash table is 64MiB (assuming no overhead), which is a similar order of magnitude to the 44MB of total L3 cache available on each of the CPUs used. As such, we can see when sufficient L3 cache capacity is available to store the entire hash table, then the workload is L3 bound, but once the hash table becomes too large, it becomes heavily external memory bound. The shift from L3 bound to memory-bound also has a clear impact on the average access time for elements, with the real-time showing substantially slower accesses for the memory-bound workloads.

However, in the benchmarks for node hash set, we can see that the benchmarks remain L3 cache bound up to table element in the range of 2^{16} , with a total data size of around 2GiB, far exceeding the L3 cache capacity, demonstrating that there is potential to more effectively use L3 cache even on large tables.

7 FURTHER WORK

7.1 Further Data Analysis

While the data presented here characterises hash tables as primarily backend and memory-bound workloads, there is a series of opportunities for further analysis using the topdown analysis. Applying a level 3 breakdown to the memory bottlenecks in high threaded may provide potential differences from the single-threaded breakdown given in figure 7 on high thread counts, which may provide more representative data on realistic large-scale compute applications.

There is also scope for a more detailed topdown analysis that targets specific elements of the code on a per hash table basis, attempting to identify key components of the code leading to branch mispredictions or cache and memory bottlenecks. This analysis would need a detailed analysis methodology able to conduct a top-down analysis only on specific regions of code in each algorithm but may provide further insights for possible optimisations. This may be particularly useful for benchmark parameters where the frontend or core are providing substantive performance limitations, as these are factors that need to be primarily resolved through improvements to the code and algorithm, rather than through microarchitectural adjustments.

The scope of benchmarks carried out in this research also is limited to just the access behaviour in the hash tables tested, so there is similar opportunities to perform detailed performance analysis

7.2 Microarchitectural Improvements

The topdown breakdown given in figure 7 indicates that on large hash tables, the table becomes substantively performance limited to external memory limitations. Therefore, more effective use of the cache has the potential to provide substantial performance increases to access on large hash tables. While part of the work to effectively use the cache needs to be placed on the programmer and compiler, there is possible scope for further scope to make changes to cache policy that provide substantial benefits to hash table based workloads.

A detailed study of the impact of microarchitectural impacts of caching size and policy is easier to conduct in a CPU simulator that allows controlled tweaking of parameters, which has a risk of introducing inaccuracies to real world simulations. However, the data collected here would allow for a detailed comparison of the effectiveness of microarchitectural simulation on the specific workloads benchmarked, and ensure simulated adjustments are representative of real-world performance behaviour.

REFERENCES

- [1] G. Zhang and D. Sanchez, "Leveraging Caches to Accelerate Hash Tables and Memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 440–452.
- [2] "Redis."
- [3] CppCon, "CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"," Oct. 2017.
- [4] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch Hashing," in *Distributed Computing*, ser. Lecture Notes in Computer Science, G. Taubenfeld, Ed. Berlin, Heidelberg: Springer, 2008, pp. 350–364.
- [5] P. Celis, P.-Å. Larson, and J. Munro, "Robin Hood Hashing (Preliminary Report)," Jan. 1985, pp. 281–288.
- [6] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

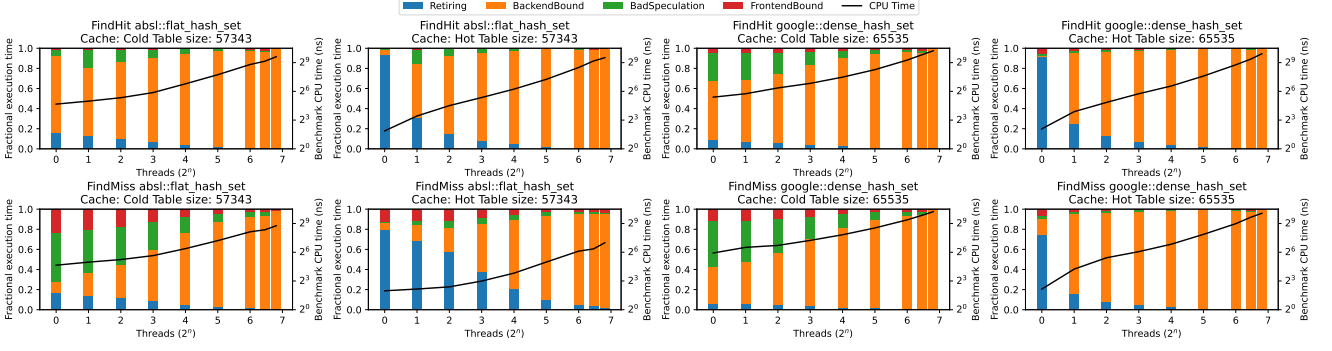


Fig. 6. Extracted data for Level 1 Top Down Performance analysis for a threaded benchmark on threads ranging from 1 to 112 threads across a range of hash tables and benchmarks. All benchmarks were carried out at maximum table density, with value sizes of 1024.

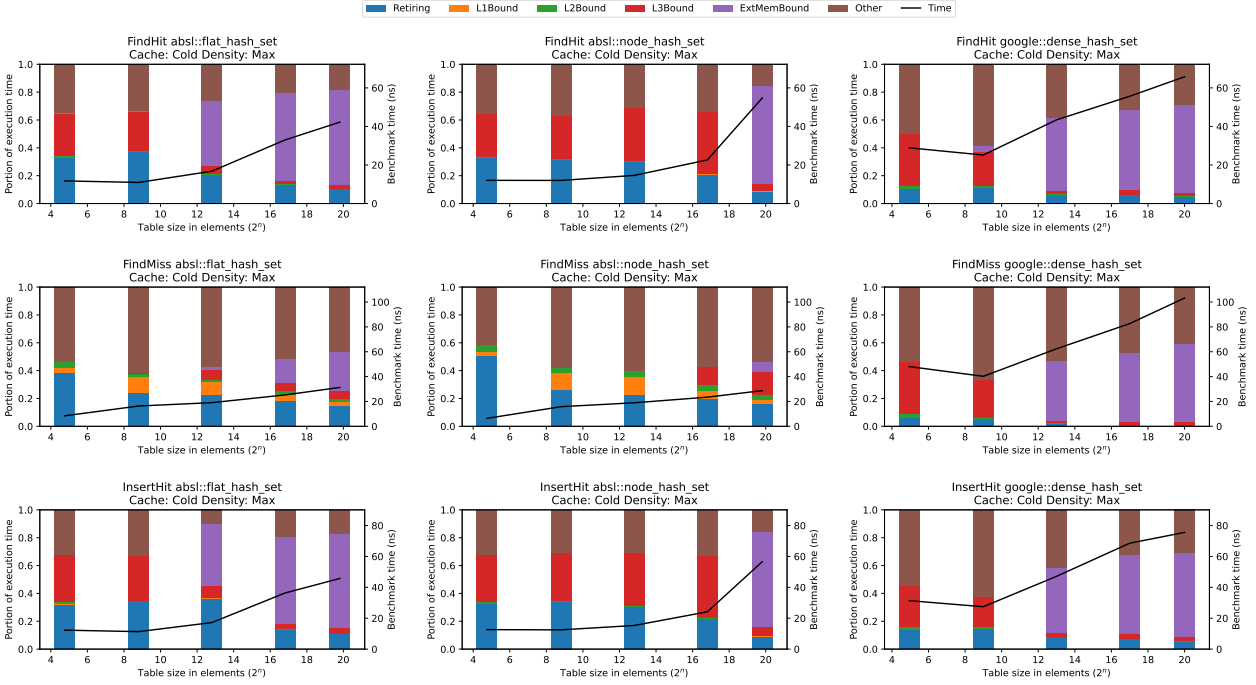


Fig. 7. Detailed analysis on memory bound portion of the top-down analysis, on a subset of parameters. All plots are for value sizes of 16384 bytes, cold cache mode and maximum table fill density.

- [7] O. Kaiyakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "SLM-DB: Single-Level Key-Value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 191–205. [Online]. Available: <https://www.usenix.org/conference/fast19/presentation/kaiyakhmet>
- [8] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 17–32. [Online]. Available: <https://www.usenix.org/conference/fast21/presentation/chen-hao>
- [9] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "ChameleonDB: a key-value store for optane persistent memory," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26–28, 2021*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 194–209. [Online]. Available: <https://doi.org/10.1145/3447786.3456237>
- [10] Y. Yang, S. R. Kuppannagari, and V. K. Prasanna, "A High Throughput Parallel Hash Table Accelerator on HBM-enabled FPGAs," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2020, pp. 148–153.
- [11] A. Yasin, "A Top-Down method for performance analysis and counters architecture," Mar. 2014, pp. 35–44.
- [12] "Abseil / Abseil Containers."
- [13] "Sparsehash/sparsehash," sparsehash, Apr. 2023.
- [14] "Hashtable Benchmarks," Google, Mar. 2023.
- [15] "Benchmark," Google, Apr. 2023.
- [16] S. Eranian and Richter, "Libpfm4 documentation."
- [17] "Top-down Microarchitecture Analysis Method," <https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html>.

APPENDIX A PERFORMANCE COUNTERS USED

The following performance counters were used to represent each top down performance component, presented in table 4.

TABLE 4
Raw performance counters

Metric	LibPFM counter
TotalSlots	TOPDOWN.SLOTS
FetchBubbles	IDQ_UOPS_NOT_DELIVERED.CORE
SlotsIssued	UOPS_ISSUED.ANY
SlotsRetired	UOPS_RETIRED.SLOTS
RecoveryBubbles	INT_MISC.RECOVERY_CYCLES
Clocks	CPU_CLK_UNHALTED.THREAD_P
MemStalls.AnyLoad	CYCLE_ACTIVITY.STALLS_MEM_ANY
MemStalls.L1Miss	CYCLE_ACTIVITY.STALLS_L1D_MISS
MemStalls.L2Miss	CYCLE_ACTIVITY.STALLS_L2_MISS
MemStalls.L3Miss	CYCLE_ACTIVITY.STALLS_L3_MISS