Reducing Hash-Table Memory Usage with Variable-Sized Tables

Arjun Rawal
Department of Computer Science
University of Chicago
Chicago, USA
arjunrawal4@uchicago.edu

***Abstract*—Dictionary data structures are an essential part of modern computer systems and algorithms. We propose a multi-tiered hash table that takes advantage of value distribution in key value stores to reduce the memory footprint of hash tables. More specifically, we fit each inserted value into a hash table that wastes the minimal amount of memory. Each hash table can be accessed and resized independently, distributing the load across multiple structures. Reducing the in-memory size of hash tables often has the side effect of improving throughput and latency, as more information can be stored in the cache. This method offers gains of nearly 50% in exponential and Zipfian data distributions, and incurs minimal latency penalties in uniform distributions. This method is easily parallelizable, applicable to different hashing algorithms, and requires no changes to existing insert, contains, and remove operations.**

***Index Terms*—Data Structure, Memory Management, Table Lookup**

I. Introduction

Dictionaries expose insert, contains, and remove operations to users, handling resizing and collisions internally. Hash collisions are particularly challenging to deal with, and a variety of strategies have been proposed to reduce the prevalence of collisions and their effect on latency and memory utilization. One fundamental problem with most strategies utilized to deal with hash collisions is memory usage. With the memory bandwidth gap growing year over year, improving data representation in memory is essential to improving performance of applications that require fast data access.

We observe that many common uses of hash tables have data that does not fully utilize the size of the datatype it is stored in. For example, a basic implementation of a hit count per website will require the allocation and storage of a datatype capable of storing the largest possible hit count per website over a year (64 bits). This memory is vastly underutilized, as a large percentage sites have hit counts that could be stored in 16 bits, and nearly every site's hit count could be stored in 32 bits. Mitigating this challenge is tricky, as allocation on demand using memory reference, and non-word aligned memory pose latency and caching difficulties, respectively. We propose disaggregating the hash table, replacing a hash table with k bits of value storage with p disjoint table, where p is on the order of $\log(k)$. Our main contributions are:

- the implementation of a hash table that reduces memory usage on values that vary in size
- an application to real-world data to reduce memory usage and data movement by up to 50%

II. Implementation

A. Base Hash Table

We implement a standard single-threaded hash table using the cuckoo hash protocol [1] as a comparison . We expose the insert, contains, and remove, operations, as well as an increment for use in counting applications. For our hash functions, we use xxHash, a throughput oriented non-cryptographic hash capable of $>$10GB/s throughput on 64 bit inputs [2]. We use two tables indexed by XXH64 functions seeded with different random values. We index using bit shifting instead of modulo to reduce the CPU cycle cost of indexing. Our implementation does naive resizing by allocating a new table and reinserting values.

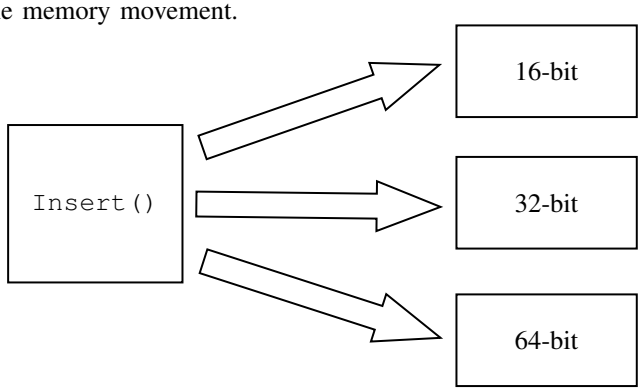
B. Adaptive Hash Table

We implement 3 tier single-threaded hash table using the cuckoo hash protocol to demonstrate the benefit of our approach. We build on top of the base hash table, maintaining all interfaces and structure, and changing only internal logic. We use 16-bit, 32-bit, and 64-bit value sizes for the three disjoint tables. This corresponds to 8, 12, and 16 byte bucket sizes, as we store the entire 32-bit key, and a byte to track the validity of an entry. We require that our entries be word aligned, and we evaluate the effect of unaligned memory. This storage format be optimized further if only 31 bits of the key are required, reducing the byte sizes to 8, 8, and 12, respectively.

When (k, v) pairs are inserted into the table, we evaluate the size of v, and then assign the insertion to the table that fits it the best. Then, that table inserts the value, and has exclusive ownership of it unless the value is changed. If the value is incremented beyond the maximum value of its current table, it is evicted and inserted into the next largest table. This protocol has several nice properties that make it memory efficient and capable of high performance.

First, each table can be accessed and resized independently, increasing the utilization of each table, and allowing for easy parallelization. Second, tables can be added or removed on demand, allowing for data to be inserted without knowing the distribution of the values or maximum value size. Finally, reducing the size of the hash table in memory has compounding

effects on throughput and latency, as more data can fit into memory, reducing the effect of cache misses and page faults. Hence, by reducing the memory size of the hash we reduce the memory movement.



III. Evaluation

A. Distributions

We use three different distributions to evaluate our results. The first distribution is uniform, representing random values in the space $(0, 2^{64})$. The second distribution is exponential, where with probability density function,

$$f(x, \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

Finally, we use a Zipfian distribution, commonly seen in linguistic and web traffic patterns. With N the number of elements, k the rank of the element in a sorted list by frequency, and s the exponent which creates the distribution.

$$f(k, s, N) = \frac{1/k^s}{\sum_{n=1}^{N} (1/n^s)}$$

Zipfian distributed values are extremely inefficiently represented in standard hash tables, as the majority of the data is clustered around a few small values. On the Zipfian and exponential distributions, we cap the maximum value at $\approx 2^{64}$. We generate random values using Python numpy, and read them in using I/O during evaluation. This slows down absolute hash table performance by approximately 30%, but should not have any effect on the relative performance of the two table models and does not change their memory usage. We considered reading the file into memory first, but found that the adverse impact of increased cache usage negatively impacted performance, and moreover, was an inaccurate model of the computation pattern, as hash tables normally operate on streaming input.

For each distribution, we consider only adding operations, as they provide the most stress to our tables, and use linearly increasing keys, as randomly shuffled keys were found to have no effect on performance. Finally, we also consider a real world application, evaluating word distribution. This task is a simple procedure to compute the frequency of words in a passage of text. We use an collection of 1.2GB of plain-text novels from Project Gutenberg archives to evaluate performance. The test function runs a set of hash functions to track word usage, shown in Algorithm 1.

Algorithm 1: Word Frequency Counter

Result: (k, v) pairs where k is a word, and v is the number of occurrences in the passage.

```
initialize(ht);
for w ∈ Text do
    if contains(ht, w) then
        increment(ht, w);
    else
        insert(ht, w, 1);
    end
end
```

B. Methodology

The tables were compiled using `gcc` version 4.8.5 with optimization -O3, and run on a 20-core Intel Xeon Gold 6138 CPU @ 2.00GHz with 27.5 MB L3 cache, and 512 GB of RAM. We initialize all three tables with 1024 slots, and grow if the cuckoo hashing policy runs more than 8 evictions. Both of these choices can be tuned for a given use case. We repeat 10 runs and take the median, to reduce the effect of outliers. We run each distribution on 10^9 hash insertions with Zipfian constant 1.2, exponential constant 0.5, and summarize distribution statistics in Table I. We also vary the constants for both the Zipfian and exponential distribution with 10^7 hash insertions, and report those results.

C. Performance Results

From Figure 1 we see that the adaptive hash table reduces memory usage of hash tables on average. On uniformly distributed inserted values, the adaptive hash table uses a slightly larger memory footprint as it allocates a 16-bit and 32-bit hash table that are not needed. However, in the exponential and Zipfian distributions, the adaptive hash reduces memory usage significantly, across all values of the Zipfian constant tested, and most of the values of the exponential constant tested (Figures 3, 5). On average, however, the memory usage for adaptive exponential should converge to no more than the standard hash, as the only excesses were from late stage resizes in one of the adaptive tables.

We also see from Figure 2 that the throughput for each distribution is relatively similar to the standard throughput. Exponential insertions are much faster, likely due to reduced cache pressure with a smaller hash table (Figure 6). Uniformly distributed values are slower as more computations have to be done for an insert, and Zipfian distributed values are more or less the same on both systems (Figure 4).

We summarize memory and throughput statistics in Table I for the tests run on 10^9 hashes.

D. Word Distribution

As seen in Figures 2 and 1, we see that for computing the distribution of words in the 1.2 GB of plain-text, the

TABLE I
DISTRIBUTION STATISTICS

Distribution	% > 2-bytes	% > 4-bytes	Hashes/ms (S)	GB of Memory (S)	Hashes/ms (A)	GB of Memory (A)
Zipfian	11.2	1.08	486	36	445	20.9
Exponential	100	13.5	569	393	442	22.1
Uniform	100	100	619	36.0	696	36.0
Words	0.04	0	16.8	0.032	21.0	0.016

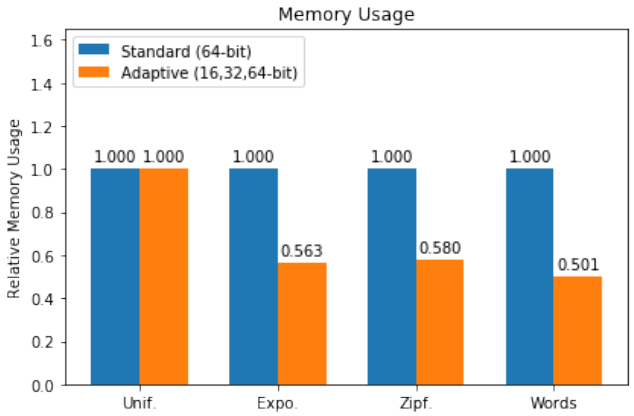


Fig. 1. Hash Table Memory Usage

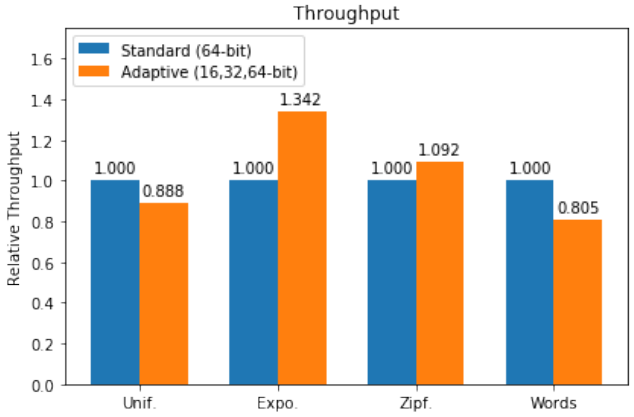


Fig. 2. Hash Table Throughput

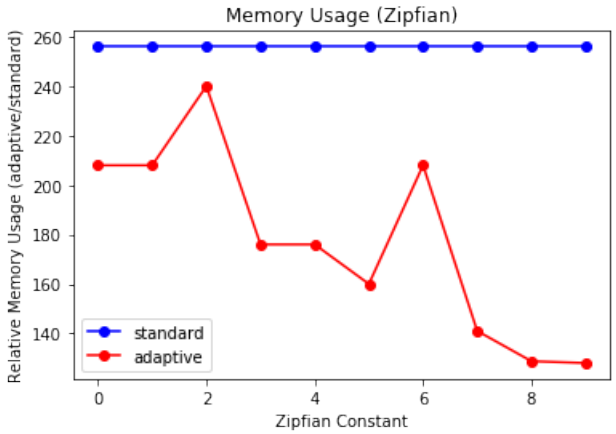


Fig. 3. Hash Table Memory Footprint (Zipfian)

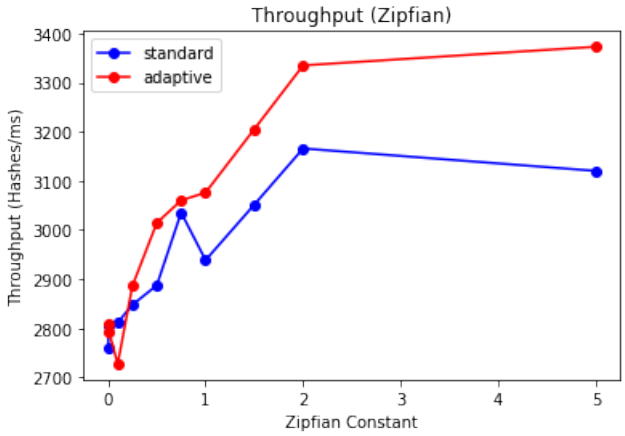


Fig. 4. Hash Table Throughput (Zipfian)

adaptive approach saves 50% of memory. This number is slightly misleading, however, as there is no reason that a 64 bit table should be used when the maximum value of a word count can be stored in a 32-bit integer. However, even when modifying the original control table to have 32-bit values, we still save 25% memory. The throughput on this test decreases slightly, likely due to the increased overhead of many calls to `contains` and `increment`, which are slower on the adaptive table.

IV. ADVANTAGES OF APPROACH

A. Application Programming Interface

The adaptive hash table maintains the existing hash table interface and requires no modification to user calls. This allows the user to insert values without having to check their size beforehand, or call different methods based on the size. This isolates the user from the internal hash table logic, and allows for simple use with the standard API calls to a hash table.

B. Extensible and Algorithm Agnostic

Although these evaluations were run using our own C implementation of a cuckoo hash table, this method can applied to any hash table that allows for rigid values. We can initialize multiple hash tables (potentially on demand), and then have a external handler that determines which table to assign an incoming value to. For contains and remove operations, there is a slowdown linearly proportional to the number of tables, but from our experiments this has a minimal effect, as the reduced cache and memory usage counteract the effect. As demonstrated in the results, when the distribution falls almost

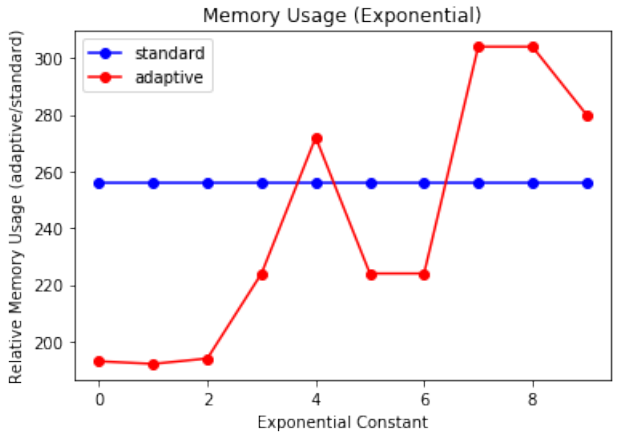


Fig. 5. Hash Table Memory Footprint (Exponential)

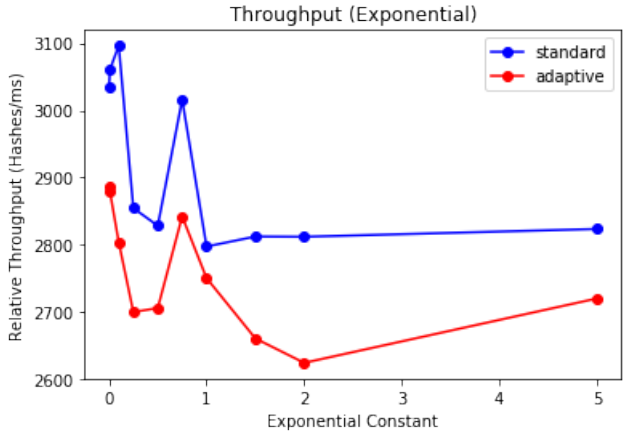


Fig. 6. Hash Table Throughput (Exponential)

exclusively within one table, the other tables remain at their initialized size.

C. Parallelization

Although the current implementation is single-threaded, there is no barrier to a parallel implementation. As mentioned in the previous section, each table can be managed independently, and can be used with any existing hash table implementation that allows control of hash table value sizing. The disaggregation of the hash table may even provide additional benefits, as the load is somewhat distributed across the different hash tables. A thread pool could be used to allow each table to request threads if needed, as the number of tables and frequency of their requests depends on the distribution of the data.

V. Summary

This presents a very limited inspection of the benefits of an adaptive hash approach to non-uniformly distributed data. In particular, data that falls on an exponential or Zipfian distribution can be stored much more compactly without significant throughput loss. Further evaluation of this approach using hash table libraries and more variations to the size and balance of add/remove calls can be done. Additionally, there is the possibility of adding memory references, and alternative compression techniques to allow adaptive hashing to store any size data without allocating very large tables.

References

[1] R. Pugh and F. F. Rodler, “Cuckoo hashing”, Journal of Algorithms Volume 51, Issue 2, May 2004, Pages 122-144.
[2] https://github.com/Cyan4973/xxHash