ADAPTIVE HASHING

Faster Hash Functions with Fewer Collisions*

Gábor Melis

melisgl@google.com

Google DeepMind

2024-05-14

* …Especially in Certain Situations

Outline

- Motivation: performance in theory and practice
- The general idea of adaptive hashing
- Adaptive `eq` hashing
- Adaptive `equal` hashing
- Wrapping up

Motivation

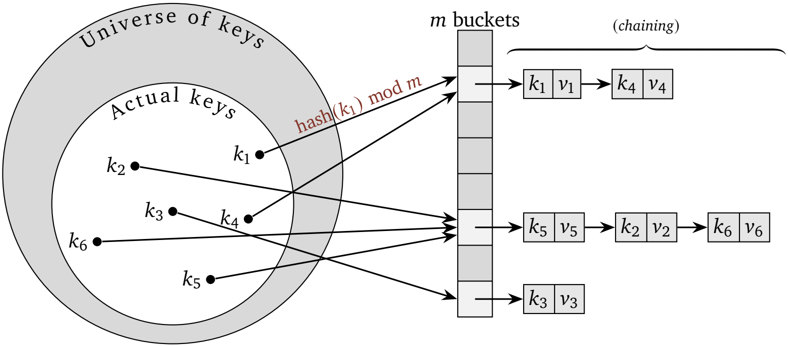
Hash tables are the most common non-trivial data structure.

- up to 50% of the time in a complex database query
- 2% of all Google CPU usage

Motivation

Hash tables are the most common non-trivial data structure.

- up to 50% of the time in a complex database query
- 2% of all Google CPU usage



Motivation: Cost and Regret in Theory

The *cost* of hashes is the expected number of comparisons for lookups.
Computed from bucket counts:

0	2	0	0	0	3	0	1

$$cost = \frac{1}{2+3+1}\left(2\,\frac{1+2}{2} + 3\,\frac{1+3}{2} + 1\,\frac{1+1}{2}\right) \approx 1.66$$

Motivation: Cost and Regret in Theory

The *cost* of hashes is the expected number of comparisons for lookups.
Computed from bucket counts:

0	2	0	0	0	3	0	1

$$cost = \frac{1}{2+3+1}\left(2\,\frac{1+2}{2} + 3\,\frac{1+3}{2} + 1\,\frac{1+1}{2}\right) \approx 1.66$$

A *perfect hash* fills buckets as evenly as possible.
They have minimal cost:

0	1	1	1	1	0	1	1

$$cost = 1$$

Motivation: Cost and Regret in Theory

The *cost* of hashes is the expected number of comparisons for lookups.
Computed from bucket counts:

0	2	0	0	0	3	0	1

$$cost = \frac{1}{2+3+1}\left(2\,\frac{1+2}{2} + 3\,\frac{1+3}{2} + 1\,\frac{1+1}{2}\right) \approx 1.66$$

A *perfect hash* fills buckets as evenly as possible.
They have minimal cost:

0	1	1	1	1	0	1	1

$$cost = 1$$

The *regret* is the cost minus the minimum achievable cost.

Motivation: Cost and Regret in Theory

The *cost* of hashes is the expected number of comparisons for lookups.
Computed from bucket counts:

0	2	0	0	0	3	0	1

$$cost = \frac{1}{2+3+1}\left(2\,\frac{1+2}{2} + 3\,\frac{1+3}{2} + 1\,\frac{1+1}{2}\right) \approx 1.66$$

A *perfect hash* fills buckets as evenly as possible.
They have minimal cost:

0	1	1	1	1	0	1	1

$$cost = 1$$

The *regret* is the cost minus the minimum achievable cost.

A *uniform hash* assigns each key to a bucket with the same probability.
0.5 expected regret at load factor 1 (eaten by $\mathcal{O}()$).

✠ There is something to gain even in theory.

Motivation: Setup for the Reality Check

The theoretical cost model is bad (duh).

Case-study on Integer Hashing:

- Keys: machine words (e.g. integer, pointer)
- Implementation: Common Lisp (SBCL)
- Comparison function: `eq` (like == in Java or `CMP` in assembly)
- Hash function: integer value / address $\longrightarrow$ hash value

We compare

- Murmur3 mixer: a general-purpose hash function ($\sim$ Uniform Hash)
- Prefuzz: SBCL's own hand-crafted `eq` hash.

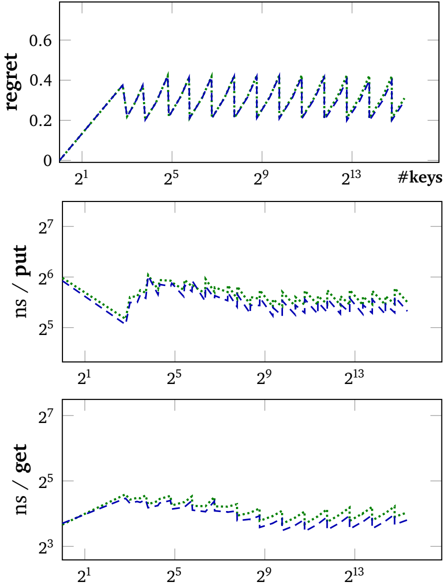
Motivation: Hash Function Speed Matters

Eq hash table performance vs the number of keys with Murmur and Prefuzz.

Keys: random existing symbol objects

Regret: Sameish. Close to a Uniform Hash.

Put, Get: Prefuzz is 5–15% faster (to compute).



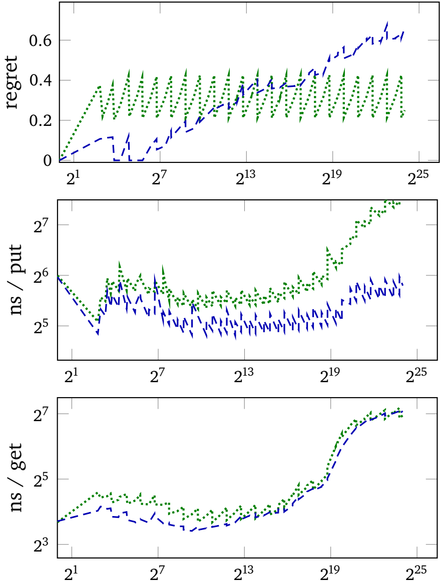
MOTIVATION: CACHE-FRIENDLINESS MATTERS

Keys: integer arithmetic progressions with increment 1 (e.g. 1, 2, 3, . . .).

Regret: Is Prefuzz optimized for small hash tables?

Put: Prefuzz is 20–75% faster due to local collisions.

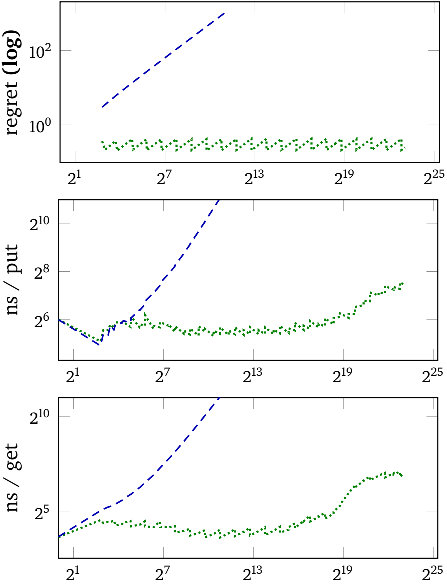
Get: Randomized query order ⇒ smaller gains



Motivation: Not Crashing and Burning Matters

Keys: single float arithmetic progressions (e.g. 1.0, 2.0, 3.0, . . .)

Prefuzz breaks.



Motivation: A Compromise Waiting to Happen

General-purpose hash functions (e.g. Murmur):

- *robust* (work with any key distribution)
- *wasteful* (do computation that doesn't improve performance)
- *suboptimal* (non-zero regret, about 0.5)
- *cache-unfriendly*

Hand-crafted hash functions (e.g. Prefuzz) are the opposite:

- *fragile* (fail outside the intended key distribution)
- *frugal* (perform minimal computation)
- can be *optimal* (zero regret)
- can be *cache-friendly*

Adaptive Hashing

Won't do:

- Perfect Hashing: static key set, offline, slow
- Dynamic Perfect Hashing: much more memory

Will do:

- Adapt the hash function to the current set of keys
- online
- to be faster
- with no change to the hash table API.

Can do fast enough?

Adaptive Hashing: Skeleton

Three low-overhead triggers for adaptation in put():

- Max chain length
- Collision count at rehash
- Hash table size

```
function put(key, value)
    bucket ← h(key) mod m
    chain_length ← 0
    for k ← next key in bucket do
        if compare(key, k) then
            value of k ← value
            return
        chain_length ← chain_length + 1
```

Adaptive Hashing: Skeleton

Three low-overhead triggers for adaptation in put():

- Max chain length
- Collision count at rehash
- Hash table size

```
function put(key, value)
    bucket ← h(key) mod m
    chain_length ← 0
    for k ← next key in bucket do
        if compare(key, k) then
            value of k ← value
            return
        chain_length ← chain_length + 1
    if chain_length too high then
        h ← safer_hash_function(h)
        bucket ← h(key) mod m
```

Adaptive Hashing: Skeleton

Three low-overhead triggers for adaptation in put():

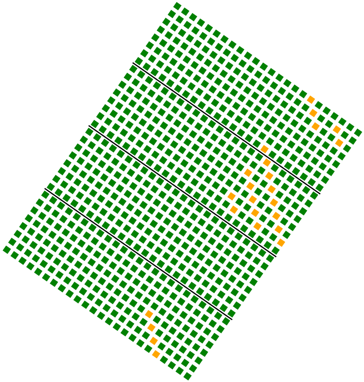
- Max chain length
- Collision count at rehash
- Hash table size

```
function put(key, value)
    bucket ← h(key) mod m
    chain_length ← 0
    for k ← next key in bucket do
        if compare(key, k) then
            value of k ← value
            return
        chain_length ← chain_length + 1
    if chain_length too high then
        h ← safer_hash_function(h)
        bucket ← h(key) mod m
    if hash table is full then
        double m and increase storage
        h ← adapt_and_rehash(h, m)
        if h was changed then
            bucket ← h(key) mod m
    add (key, value) to bucket
```

Adaptive Eq

More concretely, for `Eq` hashing:

1. Init to *Constant hash*: linear search in a vector internally
2. → *Pointer-Shift* above 32 keys
3. → *Prefuzz* if doing badly
4. → *Murmur* similarly



Adaptive Eq: Page-based Memory Allocation

Memory addresses of objects are unique.

Allocators grab a contiguous memory range from the OS (expensive).

Then, they cram many small objects into these “pages”.

Most allocations are just a pointer bump (cheap).

✠ Addresses resemble arithmetic progressions within pages.

Adaptive Eq: The Arithmetic Hash

Arithmetic progressions with odd increments are perfect hashes in power-of-2 hash tables (coprimes).

Let s be the number of low bits which are the same in all keys.

✠ $k \to k \gg s$ is a *perfect hash* for all arithmetic progressions.

Adaptive Eq: The Arithmetic Hash

Arithmetic progressions with odd increments are perfect hashes in power-of-2 hash tables (coprimes).

Let s be the number of low bits which are the same in all keys.

✠ $k \rightarrow k \gg s$ is a *perfect hash* for all arithmetic progressions.

Computing s is cheap:

function count_common_prefix_bits($k_1, \ldots, k_n$)
  mask $\leftarrow 0$ ▹ Changed bits detected so far
  for $i \leftarrow 2$ to n **do**
    mask $\leftarrow$ *mask* $\vee (k_1 \oplus k_i)$ ▹ One `OR` and one `XOR` instruction
  return count_leading_zero_bits($\neg$*mask*) ▹ A single `LZCNT` instruction

Adaptive Eq: The Pointer-Mix Hash

Multiple pages: less regular allocation patterns

Keep the low bits intact, and mix in the page:

$$\text{pointer\_mix}(k) = k \gg s \oplus \text{uniform\_hash}(k \gg n\_page\_bits)$$

For random subsets of arithmetic progressions, Pointer-Mix

- is a Perfect Hash with all keys on a single page;
- behaves like a Uniform Hash with more pages.

Adaptive Eq: The Pointer-Shift Hash

Pointer-Mix is easy to analyse but slow due to uniform_hash().

Mix in the page faster → Pointer-Shift:

```
address >> s +             /* Remove the constant low bits */
address >> n_page_bits     /* Mix in page address          */
```

Extremely aggressive, but it has Prefuzz as a safety net.

Adaptive Eq: The Prefuzz Hash

Stock SBCL's eq hash is Prefuzz:

```
address ^ 0xdeadbeef +  /* Destroy some regular patterns */
address >> 1 +          /* Mix the low bits a bit        */
address >> 4 +
address >> 13 +
address >> 21           /* Ignore the high bits          */
```

So that's why it worked well until it didn't: it punts on the high bits.

Plays it safer than Adaptive but still needs Murmur as a safety net.

✠ Similar to Pointer-Shift but does not depend on the other keys!

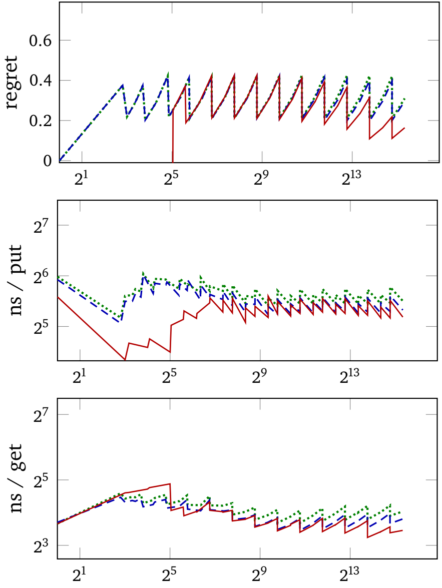
Adaptive Eq: Faster Hashing

Keys: random existing symbol objects (revisited)

Regret: Sameish. Not plotting the Constant phase.

Put: Big win for Adaptive in the Constant phase.

Get: A few percent faster than Prefuzz, which is already better than Murmur due to being a faster hash function.



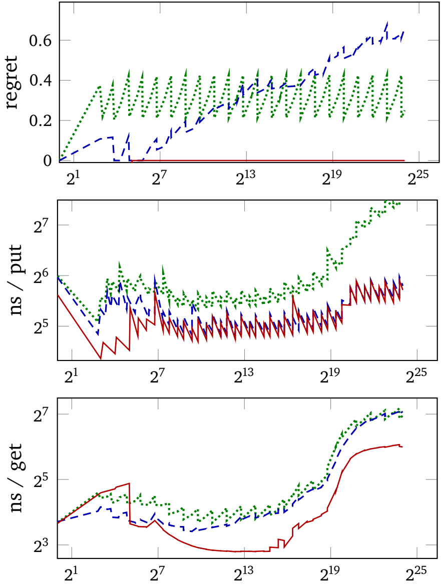
Adaptive Eq: Less Regret

Keys: integer arithmetic progression (revisited)

Regret: Adaptive is a perfect hash.

Put: 50% over Murmur, and over Prefuzz in the Constant hash phase

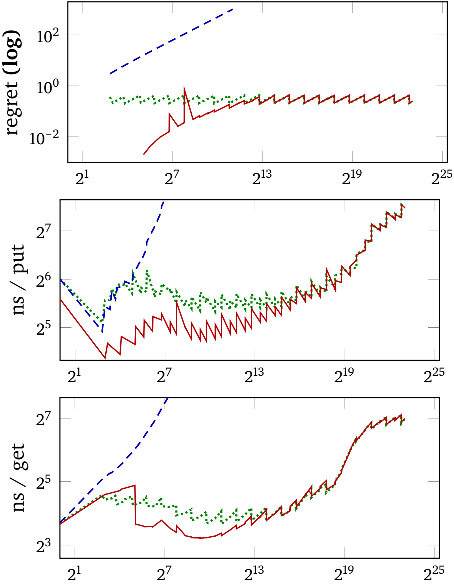
Get: Up to 50% faster



Adaptive Eq: More Robustness

Keys: single float arithmetic progressions (revisited)

Adaptive takes advantage of the regularity. When its guessed shift becomes incorrect, it falls back to Prefuzz (a bad idea) and then immediately to Murmur.



Adaptive Eq: Made in Heaven

FASTER THAN A HAND-CRAFTED HASH

Adaptive Hash

AS ROBUST AS A UNIFORM HASH

Adaptive Equal Hashing

For composite keys, running the hash function is the main cost.

- For string keys, hash only the first and last 2 characters.
- For list keys, only hash the first 4 elements.
- Double the limit if hashes are not distributed nicely.

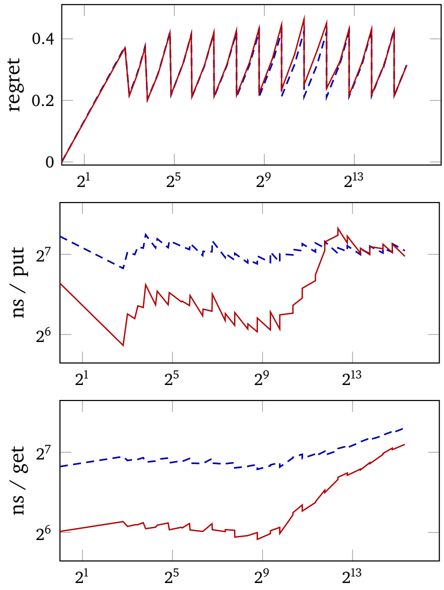
Adaptive Equal: String Keys

`Equal` hash table performance with Adaptive and SBCL.

Keys: existing strings

Regret: sameish

Put, Get: Adaptive is 30–50% faster until the truncation limit is increased beyond the length of most keys to avoid collisions.



Macrobenchmarks

Verify that the gains survive the transition to macrobenchmarks (code complexity, cache pressure).

Benchmarks:

1. compile and load a set of libraries;
2. run the tests of the same set of libraries;
3. run each test file in SBCL's `tests/` directory.

All light on hash table ops, so there is not much to gain.

✠ The relative gains are similar to those in microbenchmarks.

Limitations

- Cheap, bad proxies for performance

  Collision count and max chain length: loose lower and upper bounds

- Hard to implement without upsetting performance gods

- Requires understanding the key distribution

  E.g. the memory allocator for Pointer-Shift

Conclusions

Gains:

- Using a general-purpose hash? – Much common-case performance
- Using a weak hash? – Robustness and some performance

Lessons:

- Hash functions must depend on the actual keys for best performance.
- Hash functions can be adapted online.
- Lots of possibilities (e.g. faster DoS-resistant hashing).

✠ Better common-case performance *and* more robustness is possible.

Thanks

- Christophe Rhodes
- Miloš Stanojević
- Andrew Senior
- Paul-Virak Khuong
- The Reviewers

Code: `https://github.com/melisgl/sbcl/tree/adaptive-hash`

Paper: `https://zenodo.org/doi/10.5281/zenodo.10991321`