Adaptive Hashing

Faster Hash Functions with Fewer Collisions*

Gábor Melis
melisgl@google.com
Google DeepMind
London, United Kingdom

ABSTRACT

Hash tables are ubiquitous, and the choice of hash function, which maps a key to a bucket, is key for their performance. We argue that the predominant approach of fixing the hash function for the lifetime of the hash table is suboptimal and propose adapting it to the current set of keys. In the prevailing view, good hash functions spread the keys "randomly" and are fast to evaluate. General-purpose ones (e.g. Murmur) are designed to do both while remaining agnostic to the distribution of the keys, which limits their bucketing ability and wastes computation. When these shortcomings are recognised, the user of the hash table may specify a hash function more tailored to the expected key distribution, but doing so almost always introduces an unbounded risk in case their assumptions do not bear out in practice. At the other, fully key-aware end of the spectrum, Perfect Hashing algorithms can discover hash functions to bucket a given set of keys optimally, but they are costly to run and require the keys to be known and fixed ahead of time. Our main conceptual contribution is that adapting the hash table's hash function to the keys online is necessary for the best performance as adaptivity allows for better bucketing of keys *and* faster hash functions. We instantiate the idea of online adaptation with minimal overhead and no change to the hash table API. The experiments show that the adaptive approach marries the common-case performance of weak hash functions with the robustness of general-purpose ones.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Information systems → Hashed file organization**.

KEYWORDS

Adaptive, data structure, hash function, hash table, Common Lisp

ACM Reference Format:
Gábor Melis. 2024. Adaptive Hashing: Faster Hash Functions with Fewer Collisions. In *Proceedings of The 24th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.5281/zenodo.10991322

*...Especially in Certain Situations

ELS'24, May 06–07, 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.5281/zenodo.10991322

1 INTRODUCTION

Hash tables [13, 20] map keys to values and are one of the most fundamental data structures. As such, their performance is of considerable interest. For example, Hentschel et al. [8] claimed in 2022 that "a typical complex database query (TPC-H) could spend 50% of its total cost in hash tables, while Google spends at least 2% of its total computational cost across all systems on C++ hash tables". At that scale, even small gains in this area can have huge impact. This work aims to provide a general framework to improve the performance of hash tables in practice. Theory mostly concerns itself with the unknown key distribution setting and the cost of key lookup abstracted as the number of key comparisons required. This approach is highly successful due to its wide applicability and being a reasonable model *asymptotically* for a certain class of hash functions. However, practice stubbornly happens in the non-asymptotic regime with particular key distributions (even concrete keys) on computers with memory caches. We propose a method to equip hash tables with the ability to change their hash function based on the keys being added to improve their overall performance by bucketing the keys more evenly, making the hash function faster or more cache-friendly. We are not aware of a cost model that incorporates these effects and is amenable to theoretical analysis, hence this work tilts heavily towards Experimental Algorithmics [14].

It is useful to contrast our method with hand-crafting hash functions for particular key distributions. We argue that hand-crafting is too rigid: it breaks badly if the distributional assumption is violated [9, 10]. It is also costly in terms of human labour, and designing hash functions with guarantees is hard. While selecting the hash function offline amortizes the design cost, it also attempts to solve a much harder problem than necessary in balancing the complexity and the quality of the hash function across all possible sets of keys.

A less rigid solution is to select a hash function *online* given the actual keys in the hash table. This pulls the hash function selection cost into the runtime realm, and we must be extremely cautious of introducing overhead lest any possible gains be wasted. Here, we propose hiding some of the selection cost in rehashing, when the hash table grows. In summary, our main contributions are:

- On the conceptual level, we argue that hash functions must be able to change during the lifetime of a hash table for best performance.
- We propose light-weight adaptation mechanism tailored to string and integer/pointer hashing.
- We empirically demonstrate performance gains in a real-life hash table implementation.[1]

[1]The proposed algorithms were implemented within SBCL [15], a high-quality Common Lisp. SBCL hash tables are reasonably fast given the constraints of the ANSI standard [19] but not close to the state of the art. We microoptimized the baseline SBCL v2.4.2 to make performance comparisons fairer.

Algorithm 1 Sketch of a possible adaptation mechanism implemented in *put* with low overhead. The unchanged parts of a usual *put* implementation are grayed out. The hash function may be adapted when there are too many keys in the same bucket, or when rehashing finds that the number of collisions is too high. Alternatively, the total cost of access (Definition 2) could be tracked, requiring an additional write to memory and also a performance penalty to key deletion.

```
1:  function put(key, value)
2:      bucket ← hash(key) mod m
3:      chain_length ← 0
4:      for k ← next key in bucket do
5:          if (hashes are not cached) or cached_hash(k) = h then
6:              if compare(key, k) then
7:                  value of k ← value
8:                  return
9:          chain_length ← chain_length + 1
10:     if chain_length too high then
11:         h ← safer_hash_function(h)
12:         bucket ← hash(key) mod m
13:     if hash tabel is full then
14:         double m and increase storage
15:         adapt_and_rehash()
16:         if hash function was changed then
17:             bucket ← hash(key) mod m
18:     add (key, value) to bucket
```

To ground the exposition, Algorithm 1 sketches the implementation of the high-level part of a possible adaptation mechanism. In later sections, we flesh out this skeletal algorithm.

2 THE CASE FOR ADAPTIVE HASHING

In this section, we present the traditional cost model for lookup in hash tables, which is based on the number of comparisons, then characterize how much being key-agnostic costs in these terms.

We denote the number of buckets with m, the number of keys with n, the keys (assumed to be integers) with k, hash values with $h \in \mathbb{N}_0$, buckets with $b \in [0, m-1]$, and say that hash h falls into bucket b if $h \bmod m = b$. We often refer to vectors of certain sizes, e.g. of hashes, with notation like $h_{1:n}$.

Definition 1 (Bucket Count)**.** *For a given set of hash values $h_{1:n}$ and m buckets, we denote the bucket count vector with $c(h_{1:n}, m) \in (\mathbb{N}_0)^m$, where $c(h_{1:n}, m)_b = |\{i\colon i \in [1, n], h_i \bmod m = b\}|$ is the number of hashes falling into bucket b, for all $b \in [0, m-1]$.*

Next, we define the cost of hash values at a given number of buckets to be the expected number of comparisons one has to make to find the value associated with a key present in the hash table. For a single bucket with c_b hashes, this is $(c_b + 1)/2$ assuming a uniform distribution over the keys being looked up.

Definition 2 (Cost of Hashes)**.** *The cost of hashes $h_{1:n}$ with m buckets is $C(c) = n^{-1} \sum_{b=0}^{m-1} c_b(c_b + 1)/2$, using the shorthand $c = c(h_{1:n}, m)$.*

Note that this definition differs from *hash function quality* in the Dragon Book [1] only in the normalization.

We define perfect hashes as those that fill all buckets as equally as possibile. This generalizes the classic definition [7], which requires no collisions, to the space-restricted setting of $m < n$.

Definition 3 (Perfect Hash)**.** *We say the hashes $h_{1:n}$ are perfectly distributed in m buckets if $m - (n \bmod m)$ buckets have $\lfloor n/m \rfloor$ hashes in them, and $n \bmod m$ buckets have $\lfloor n/m \rfloor + 1$ hashes in them.*

Proposition 4. *[Perfect Hashes have Minimal Cost] Let $U(n, m)$ be the bucket count vector of any perfect hash of n keys and m buckets. Let $q = \lfloor n/m \rfloor$ and $r = n \bmod m$. Then,*

$$C(U(n, m)) = (m - r)\frac{q(q+1)}{2n} + r\frac{(q+1)(q+2)}{2n},$$

and this cost is minimal.

For the proof of this and other propositions, see Appendix A.

In the illustrative special case of an integer load factor n/m, we have that $n = qm$ (i.e. $r = 0$), the counts will be the same for all buckets, and $C(U(n, m)) = m\frac{q(q+1)}{2n} = \frac{q+1}{2}$.

Since perfect hashes have minimal cost, we define the regret of a hash the excess cost over that.

Definition 5 (Regret of Hashes)**.** *The regret of hashes $h_{1:n}$ is*

$$R(h_{1:n}, m) = C(c(h_{1:n}, m)) - C(U(n, m)).$$

Note that the classic cost model in Definition 2 is simple but clearly wrong if hashes of keys are cached (see Algorithm 1) because it costs one memory access to look up the cached hash for a key, but the cost of key comparison may be much higher. Still, with random hashes of many bits, this distinction becomes moot for regret because only one cached hash is likely to match, so there will be a single comparison for each lookup, and their contributions to $C(c(h_{1:n}, m))$ and $C(U(n, m))$ cancel out. Thus, with cached hashes, the regret can be interpreted to be in terms of memory access.

Hash functions strive to be indistinguishable with reasonable effort from a uniform hash [5], which assigns each key to a bucket with uniform probability, whose cost we consider next.

Proposition 6. *[Expected Cost of the Uniform Hash] Let P be a uniform distribution over functions that map keys to buckets. Then,*

$$\mathop{\mathbb{E}}_{\pi_{1:n} \sim P} C(c([\pi_1(k_1), \ldots \pi_n(k_n)], m)) = 1 + \frac{n-1}{2m},$$

where $\pi_{1:n}$ are n independent samples from P.

The uniform hash is optimal among hashes that are functions of a single key. However, its cost is clearly worse when compared to a perfect hash, which can be viewed as having knowledge of all keys. Next, we characterize its regret, assuming that $m \mid n$, for convenience.

Proposition 7. *[Expected Regret of the Uniform Hash] For all load factors $q \in \mathbb{N}$ ($n = qm$), the expected regret of the uniform hash is $0.5 + \frac{1}{m}$.*

So, the uniform hash needs about one extra comparison per two lookups compared to a perfect hash because it does not take all the keys into account. That's not good but not terrible either. It may be worth improving, especially if the hash function can be made faster at the same time.

Algorithm 2 Hashing strings with a *limit* on the number of characters taken into account. The algorithm moves inwards from the two ends of the string because those tend to be the most informative and because this scheme can be easily extended to reuse a previously computed hash with a lower limit. The function *add_char* performs one step of the FNV-1A algorithm.

```
1:  function hash_string(s, limit)
2:      h ← len(s)                          ▷ Initialize the hash to the length
3:      a, b ← 0, len(s) − 1
4:      n ← min(limit, l)
5:      while a < (n ≫ −1) do
6:          h ← add_char(h, s[a])
7:          h ← add_char(h, s[b])
8:          a, b ← a + 1, b − 1
9:      if n mod 2 = 1 then                 ▷ Add the odd middle character
10:         h ← add_char(h, s[a])
11:     return h
```

3 HALF AN EXAMPLE

In related work, Hentschel et al. [8] engage with one half of this problem: they select high-entropy parts of the key to feed to a general-purpose hash function. Their approach can speed up the hash function but cannot reduce the expected number of collisions. Crucially, once a hash function has been learned in an offline manner for a given key distribution, it remains fixed for the lifetime of the hash table. In a similar vein but adapting the hash function on the fly, we demonstrate significant speedups on string hashing even with slightly more collisions.

In particular, we hash only a subset of the data in compound keys, where the size of the subset is subject to a dynamically adjusted limit. In case of string keys, we limit the number of characters hashed. Hashing proceeds inwards alternating between taking a character from the beginning and the end of the string. The algorithm (FNV-1A [21]) is initialized with the length of the string to cheaply introduce some information about the truncated away characters into the hash. See Algorithm 2 for the code listing.

To detect overly severe truncation, we track the maximum chain length. That is, when a new key is being inserted whose hash is computed with truncation (e.g. it's a string longer than the current limit), we check the number of keys already in its bucket. If the probability of that many keys having collided with the uniform hash (without truncation) is less than 1%, we double the limit and rehash. Since hashes of strings are expensive to compute, they are cached (see Algorithm 1, Section 2, and Appendix C). By compromising the hash function's quality, we run the risk of having to perform more comparisons, which can be costly, thus, it is important to have a tight limit on the chain length, which we achieve by precomputing them for all possible power-of-2 bucket counts at load factor 1 and changing the current limit when the hash table is resized.

If after this rehash, the number of collisions is significantly higher than would be expected with the uniform hash, then we double the limit again and rehash. This procedure repeats until there are no more keys with truncated hashes[2] or the number of collisions falls near the expected level (see Appendix B).

[2]We use the highest bit in the hash to indicate truncation.

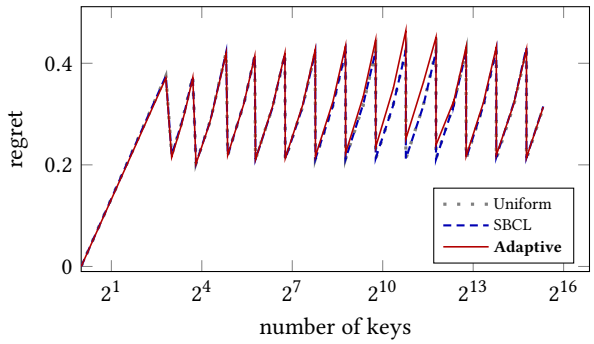

Figure 1: Regret (Definition 5) with string keys. Adaptive does not gain or significantly compromise on regret. Points where the truncation limit changes vary between runs.

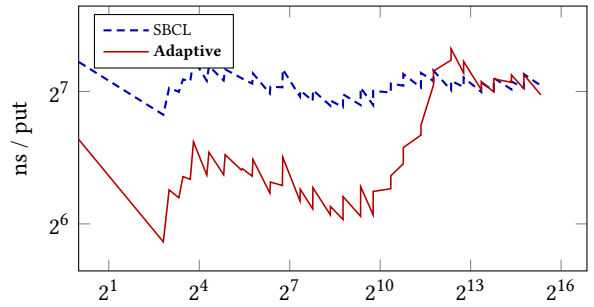

Figure 2: PUT timings in nanoseconds with string keys. Note the log scales. The plot shows the *average* time for inserting a new key when populating an empty hash table with a given number of keys.

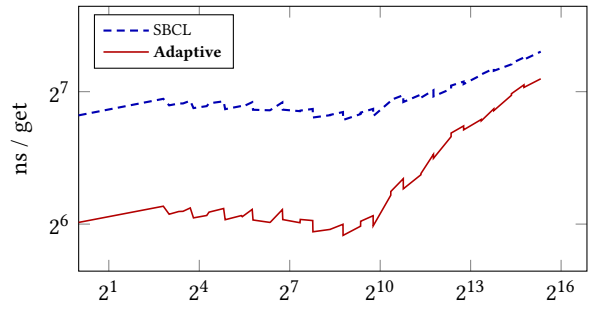

Figure 3: GET timings with string keys.

3.1 Experiments with String Keys

We implemented the adaptive hash algorithm by changing SBCL's standard `equal` hash tables[3]. Then, we collected all different strings present in the running Lisp, which gave us about 40 000 keys and measured the time it takes to populate hash tables from an empty state, averaging over same-sized random subsets of the keys. We partitioned the range of possible key counts into maximally large segments within which the hash table internal data structures are

[3]Common Lisp's `equal` is like Java's `.equals()`: it compares two objects by value.

not resized and measured performance with the lowest and highest possible key count in each segment. We also measured the time it takes to look up an existing key (GET), to look up a key not in the hash table (MISS), and to delete an existing key (DEL). All reported times are in nanoseconds per operation (e.g. insertions for populating the table, lookups for GET). For details of the experimental setup see Appendix F.

Figures 1 to 3 show our results. The jaggedness of the lines is the effect of hash table resizing. At smaller sizes, the gains are considerable and similar to those in Hentschel et al. [8]. With the current implementation, the performance of the adaptive method eventually falls back to the baseline (unmodified SBCL) because the max-chain-length check in Algorithm 1 gets triggered by strings that share a long common prefix and suffix, pushing the truncation limit beyond the length of most keys.

Note that a more advanced implementation could reduce the overhead of rehashing by starting Algorithm 2 from the hash value produced with a lower *limit* and only hashing the characters beyond that. While this would help PUT results a bit, GET would not benefit. The more fundamental problem is that max-chain-length is a really loose indicator of the cost, and we should track the regret instead.

3.2 Experiments with List Keys

The solution used in the string case also works for lists with a small modification, which is another common type of key in `equal` hash tables. Since lists are not random-access, we only consider their prefixes (unlike Algorithm 2, which also includes suffixes of strings). At least since the year 2000, stock SBCL has truncated list keys to length 4 to avoid stack exhaustion in case its recursive hashing algorithm is invoked on a circular key. This value might have been chosen empirically to maximize performance, or user code has adapted to this limitation even if the root cause of the issue remained unrecognised, or both. Regardless, we found that a default limit of 4 worked best. From this default, the limit is increased as collisions warrant, as in the string case. So, the practical gains for the adaptive method may be limited to cases where crucial bits in keys are put unknowingly beyond length 4. Curiously, there are two such cases in SBCL's own test suite (`arith-combinations.pure.lisp` and `save4.test.sh`), which experience a disastrous number of collisions and are sped up by 60% when the adaptation mechanism increases the truncation length.

4 INTEGER AND POINTER KEYS

Section 2 indicates that it may be possible to improve the regret by adapting the hash function to the keys on the fly, but whether there is a practical implementation of adaptation – which covers a non-trivial set of workloads and is lightweight enough to benefit overall performance – remains to be demonstrated.

In the previous section, we showed an example of how to reduce the complexity of the hash without increasing the number of collisions too much. In this section, we instantiate the general idea of adaptive hashing on the problem of hashing integer and pointer keys [11]. In this case, we will consider reducing the number of collisions at the same time as speeding up the hash function, requiring more than a passing familiarity with the key distribution.

4.1 Perfect Hashing on Arithmetic Sequences

First, we consider the idealized case of adding keys to a power-of-2 hash table from an arithmetic sequence of integers in order. Let $a_i = a_0 + id$ for all $i \in [1, \dots]$, where a_0 is the offset and d is difference between successive elements. a perfect hash here is $h_i = \lfloor a_i/d \rfloor = h_0 + i$, but this requires division by an arbitrary constant d, which is slow on current hardware.

Since the number of buckets m is a power of 2, as long as d is odd, any finite progression in a will be perfectly distributed modulo m because d and m are coprime. Let s be the largest integer such that $2^s \mid d$. Then $h_i = \lfloor a_i/2^s \rfloor$ is an arithmetic sequence with odd increment $d/2^s$, thus perfectly distributed. So, if we know s, we can use the perfect hash function $k \to k \gg s$ with a single arithmetic shift. Less regular than arithmetic sequences are the addresses of sequentially allocated objects, which we consider next.

4.2 Page-Based Memory Allocators

If keys are memory addresses (e.g. pointers to objects), then we may be able to take advantage of how the allocator works. We consider the case of page-based allocators, which first allocate contiguous memory ranges called pages from the OS. From these pages, they are then able to allocate objects much more quickly. To decrease contention, pages are often assigned to individual threads. A thread may have multiple pages assigned to it, in which case it may choose between pages based on the allocation size. In particular, TCMalloc allocates pages of 8KB by default and has allocations of roughly the same size within the same page. SBCL, a Common Lisp implementation with a moving garbage collector (GC), has two 32KB pages per thread: one for conses (whose size is two machine words), and another for all other objects[4].

Under such allocators, if the hash table keys are of the same size and are allocated in a tight loop, we can expect to have roughly arithmetic progressions in terms of addresses. But only roughly because with TCMalloc there may be holes (that belonged to previously freed objects) on the page, which may be filled in a more irregular pattern, and when a page is full, the new page may be anywhere in memory. With SBCL, pages have no holes because allocation within a page is simply a pointer bump[5] and because the GC compacts. When there is not enough room left on the current page or GC happens, the allocator gets a new page, but the addresses of subsequent pages are much less regular than the address within pages. A further complication with SBCL is that there are no separate pages for objects of different sizes, so if objects of non-constant sizes are allocated between subsequent keys, that throws regularity off and may reduce the density of addresses of keys within the page.

In summary, to the extent that addresses of keys are distributed like elements of pure arithmetic progressions, their hashes can be improved. We leverage the following properties:

- **denseness**: many keys are allocated on the same page,
- **alignment**: the power-of-2 alignment of keys is constant (especially within a single page).

[4]This is a simplification. Some platforms also have immobile space, arenas, and "large" objects are handled specially.

[5]The address of the next object is the address of the previously allocated object plus its size aligned to a double word boundary.

Algorithm 3 Detecting common low bits in integer keys (e.g. pointers from page-based allocators) $k_1, \ldots k_n$. This is to find the largest power-of-2 factor of the common difference in an arithmetic progression regardless of the offset caused by the first term. The symbols $\vee$, $\oplus$, $\neg$ denote the bitwise OR, XOR and NOT operations. Note that *count_leading_zero_bits* is often a single assembly instruction such as LZCNT on x86.

```
1: function count_common_prefix_bits(k_1, ..., k_n)
2:     mask ← 0                          ▷ Changed bits detected so far.
3:     for i ← 2 to n do
4:         mask ← mask ∨ (k_1 ⊕ k_i)
5:     return count_leading_zero_bits(¬mask)
```

4.3 Detecting Common Power-of-2 Factors

We need to detect s in the factor 2^s common to all keys fast and safely. Fast because this will be done at runtime, and safely because using an overestimation of s in e.g. the Arithmetic hash $k \to k \gg s$ can discard valuable bits and lead to a disastrous number of collisions, which must then be detected and corrected by the another change to the hash function (see Algorithm 1 and Algorithm 4).

Algorithm 3 is designed to fulfil these requirements. It is extremely light, performing about 2 bitwise assembly instructions per key, over only a subset of keys to limit memory access. Also, it detects the common factor without assuming that the sequence is arithmetic, which makes it applicable in more circumstances.

As to safety, if we have n keys, then the probability of a bit appearing constant by chance is 2^{1-n} (assuming that bits are Binomial(0.5) in the hash values). Thus, in practice, we can detect constant low bits with high probability with as few as 8-16 keys. We use the detected shift s in the following three hash functions.

4.4 The Arithmetic Hash

The Arithmetic hash function is $k \to k \gg s$, where $s \in \mathbb{N}_0$. As discussed in Section 4.1, this is a perfect hash function for arithmetic progressions.

4.5 The Pointer-Mix Hash

The Arithmetic hash function can easily have high cost if, for example, pointers on multiple pages come from the same smallish subset on each page.

Our next hash function, Pointer-Mix, combines the Arithmetic hash $k \gg s$ with a general purpose hash of the page address $k \gg \text{PB}$, where PB is the base 2 logarithm of the allocation page size in bytes. The Pointer-Mix hash function is $k \to k \gg s \oplus \text{safe}(k \gg \text{PB})$, where $\oplus$ is the bitwise XOR operation, and safe() is a general purpose hash function such as Murmur3.

Next, we characterize its regret in the setting discussed in Section 4.2, when keys are allocated in a tight loop but do not fit on a single page. The keys are pointers to objects distributed uniformly between multiple pages and within pages form subsets of values of arithmetic sequences of the same increment.

Proposition 8. *[Expected Cost of Pointer-Mix] Let $k_{1:n}$ be integer keys, and $\mathcal{P} = \{k_i \gg \text{PB} : i \in [1, n]\}$ the set of pages (the high bits of keys). Let the keys be distributed over the pages uniformly, $n = |\mathcal{P}|u$, where u is the number of keys on the same page ($u = |\{i \colon k_i \gg \text{PB} = p\}|$ for all pages $p \in \mathcal{P}$). We assume that all u keys on the same page form random subsets of arithmetic progressions with page specific offsets but the same increments. Then, the expected cost of the Pointer-Mix hash function is*

$$1 + \frac{n - u \min(1, \frac{2^{\text{PB}-s}}{m})}{2m}.$$

See Appendix A for the proof.

This cost is upper bounded by that of the uniform hash (Proposition 6), $1 + \frac{n-1}{2m}$, and we can see that with a few densely packed pages, we can get reasonable improvements, which diminish quickly with more pages and sparsity. Meanwhile, at the single-page extreme ($u = n \leqslant m$), Pointer-Mix is a perfect hash.

4.6 The Pointer-Shift Hash

We have seen that as the number of pages grows, Pointer-Mix quickly falls back to the performance level of the uniform hash. It is also slow: it includes a general purpose hash.

In practice, we found that the Pointer-Shift hash $k \to k \gg s' + k \gg \text{PB}$ often outperforms Pointer-Mix. Furthermore, it also behaves rather similarly to the Arithmetic hash if s' and PB are not close in value. To avoid degenerating to $k \to 2k \gg \text{PB}$ at $s = \text{PB}$, s' is set to a large value in this case to zero out the first term without introducing a slow conditional.

4.7 The Constant Hash

The hash functions discussed up to now are adaptive as they all involve the shift s detected from the keys. A different kind of adaptation, based the number of keys but ignoring their values is also possible. We mirror the common practice of starting with an array plus linear search and switching to a hash table above a predefined, small number of keys but hide it behind the hash table API and utilize it when the comparison function is extremely light as is the case with integer / pointer hashing. This may also be viewed as as a *Constant* hash with a specialized implementation or a hash table with only one bucket.

4.8 Other Hashes

Stock SBCL comes with the *Prefuzz* hash function, which was designed by hand and performs well empirically in many situations. Naturally, Prefuzz takes advantage of the memory allocation patterns to make common use-cases fast (e.g. `symbol` keys, frequently used in the compiler), but it does so at the expense of extreme penalties to others.

The *Murmur3* mixer function is a fast, widely used, general-purpose, non-cryptographic hash function with strong mixing properties. Its bucket distribution is very close that of the uniform hash.

4.9 Adapting the Hash Function

We implemented the above hash functions in SBCL and modified its hash table implementation to perform adaptation with pointers and integers. So, we use Common Lisp's `eq` hash tables, whose comparison function is based on object identity (i.e. it compares the address of non-immediate objects, or the value of immediate objects such as integers that fit into a machine word) similarly to the == operator in Java.

Algorithm 4 Adapting the hash function in `eq` (i.e. object identity based) hash tables at rehash. Note that we count collisions with Prefuzz only at larger sizes; otherwise it adapts only through the max-chain-length mechanism (see Algorithm 1) to reduce the overhead. We refer to this algorithm as Co+PS>Pr>Mu when comparing minor variations.

Require: Integer/pointer keys $k_i (i \in [1, \ldots, n])$, doubled number of buckets m, current hash function h.

```
1:  procedure adapt_and_rehash_eq
2:      if h = constant_hash then
3:          if m = 64 then
4:              s ← count_common_prefix_bits(k_{1:10})
5:              h ← pointer_shift
6:      if h = pointer_shift then
7:          n_collisions ← rehash(m, h, count_collisions = True)
8:          if n_collisions is too many then
9:              h ← prefuzz
10:     if h = prefuzz then
11:         if m < 2048 then
12:             rehash(m, h)
13:         else
14:             n_collisions ← rehash(m, h, count_collisions = True)
15:             if n_collisions is too many then
16:                 h ← murmur3
17:     if h = murmur3 then
18:         rehash(m, h)
```

The adaptation mechanism for `eq` hash tables (Algorithm 4) fleshes out Algorithm 1 and works as follows. Hash tables are initialized with the Constant hash, which remains in effect until the number of keys exceeds 32. At that point, *count_common_prefix_bits* determines the shift s to use, and we switch to the Pointer-Shift hash function. Whenever rehash finds that there are significantly more collisions than would be expected with a uniform hash (see Appendix B), the hash table switches to Prefuzz.

We also need to fall back on Murmur3 in case Prefuzz produces too many collisions. However, because Prefuzz is somewhat robust and considerably faster than Murmur3, we count collisions only at larger sizes to reduce the adaptation overhead[6] and otherwise rely on the max-chain-length mechanism from Algorithm 1 to fall back on the next safer hash function (in the order they appear in *adapt_and_rehash_eq* in Algorithm 4) if the key being inserted falls into a bucket with at least 14 other keys[7]. Using the tighther limit from Section 3 would be too costly for `eq` hashing. See Appendix D for the more details.

In summary, we have two ways of detecting when the current hash function is likely suboptimal: tracking collisions at rehash and chain length at insertion. We use collision tracking to catch gradual degradations of performance, and max-chain-length to catch catastrophic failures in a single bucket. One can construct lower and upper bounds on the average cost of lookup based on the collision count and max-chain-length. Neither of these mechanisms are perfect, but they work quite well in tandem to inform the adaptation logic about the cost of lookup.

[6]The overhead is that of incrementing a single counter if the bucket in `index-vector` is not zero, which is a hard to predict branch for the CPU.

[7]With a uniform hash, the probability of the maximum chain length being at most 14 is higher than 99% for all possible hash table sizes.

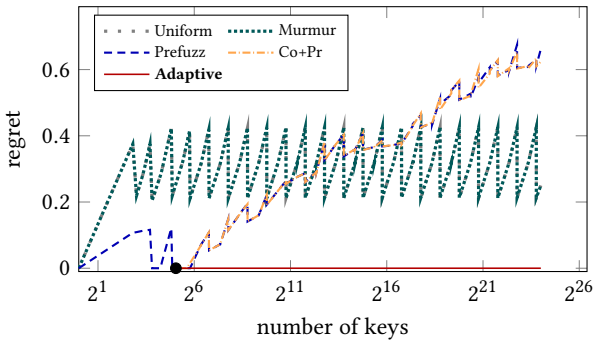

Figure 4: Regret with `FIXNUM :PROG 1`. Murmur closely tracks Uniform. Prefuzz is aggressively optimized for small sizes. Adaptive (Algorithm 4) is a perfect hash here. Both Co+Pr (Constant followed by Prefuzz) and Adaptive use the Constant hash until the fixed switch point at 32 keys (black dot).

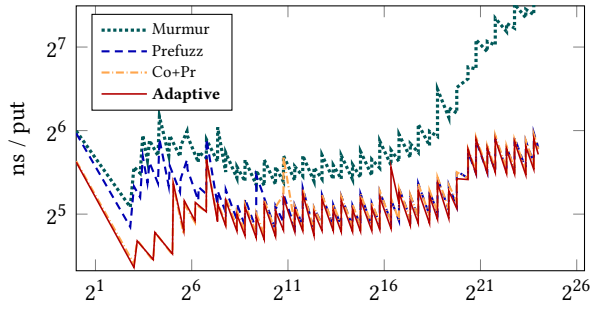

Figure 5: PUT timings with `FIXNUM :PROG 1`. Prefuzz outperforms Murmur even at large sizes despite higher regret because it's friendlier to the cache (its collisions are between subsequent elements of the progression), and its combination with Constant is even faster. Thus, despite being a perfect hash, Adaptive can improve on them only marginally.

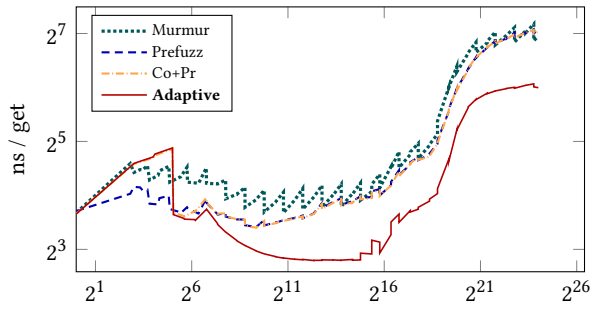

Figure 6: GET timings with `FIXNUM :PROG 1`. Keys are queried in random order so regret matters more here than with PUT, but the cache-friendliness of Prefuzz still keeps it ahead of Murmur. As expected, Adaptive can finally benefit from its zero regret after the Constant hash phase.

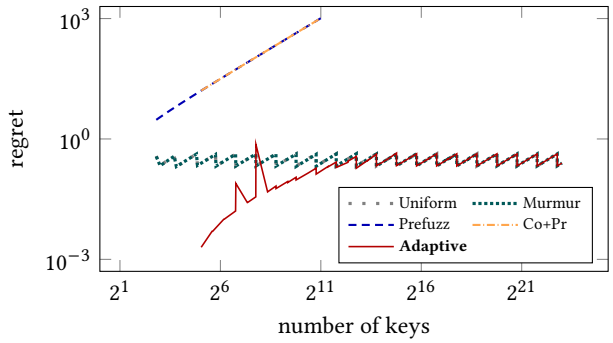

Figure 7: Regret with `FLOAT :PROG 1`. To be able to plot the catastrophic failure of Prefuzz (and of Co+Pr, consequently), we use log scale for regret on this graph. Single floats are especially problematic for Prefuzz because they can have many constant low bits. Adaptive detects these constant low bits and does better than Uniform until variation in the floating point exponents makes its original estimate of the number of constant low bits invalid, and the resulting gradual increase in collisions makes it switch to Prefuzz at rehash time. This is a spectacularly bad idea in this scenario, and the high number of collisions causes an immediate switch to Murmur. The switch times vary by hash table because the key sets are generated starting from random offsets.

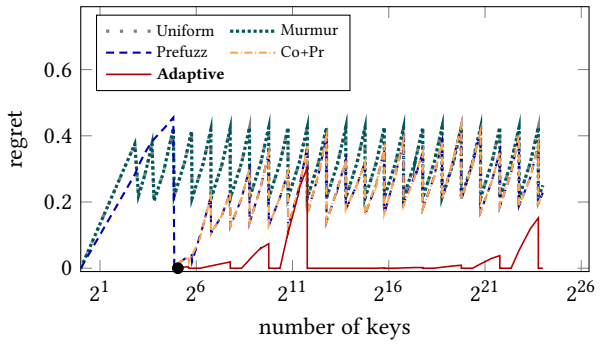

Figure 8: Regret with `FIXNUM :PROG 12`. Murmur closely tracks Uniform, but Prefuzz is better across almost the whole range. Arithmetic (Section 4.4) would be a perfect hash here, but Adaptive, which uses Pointer-Shift (Section 4.6), is not quite perfect due to the interference of its $k \gg$ PB term.

Both the collision count and max chain length are proxies for the cost of lookup as defined in Definition 2, which in turn is a proxy for performance that ignores important factors such as cache effects. Thus, even if adaptation can be driven by proxy statistics, there is no way around actually measuring performance directly.

4.10 Microbenchmarks

We conducted experiments on SBCL's `eq` hash tables with various hash functions, hash table sizes, integer and pointer keys. Our experimental methodology is the same as in Section 3.1 except for how keys are generated, which we briefly describe below (see Appendix F for the details). In the experiments, Co+Pr starts with the Constant hash and switches to Prefuzz above 32 keys. Adaptive behaves as described in Algorithm 4.

- (`FIXNUM :PROG 1`) The first experiment is with `fixnums`[8] following an arithmetic progression with increment 1 (denoted by `:PROG 1`). As Figure 4 shows, the regrets of Murmur and Uniform are very close, as expected. Prefuzz seems to be aggressively optimized for small hash table sizes, and Adaptive is a perfect hash in this simple scenario. However, differences in regret do not predict actual performance well, which is most visible in Figure 5, where insertion is much faster with Prefuzz than with Murmur even where the latter has much smaller regret. This is because the collisions with Prefuzz are between keys close in insertion order, which benefits the CPU's cache. The same effect is present in lookups (Figure 6), although to a lesser degree because the benchmark looks up keys in random order, so some random access to memory is inevitable. See Appendix I for the full set of results.
- (`FLOAT :PROG 1`) Similar to the previous `fixnum` case, we also tested `single-float` keys. As the regret curves in Figure 7 show, Prefuzz suffers a catastrophic failure, placing most keys in the same bucket, but Adaptive takes advantage of the many constant low bits in the keys, eventually falling back to Prefuzz and then to Murmur. See Appendix J for the full set of results.
- (`FIXNUM :PROG 12`) Next, we tested arithmetic progressions with increment 12. This is intended to test whether the shift detection in Algorithm 3 works. The regret curves in Figure 8 and the operation timings tell a similar story as with `:PROG 1` except that Adaptive is no longer a perfect hash due to Pointer-Shift's $k \gg$ PB term. See Appendix K for more.
- (`FIXNUM :RND 6`) Like `:PROG 6`, but keys follow a random progression: 0–5 values are skipped randomly between subsequent keys. This is intended to approximate populating a hash table with non-uniformly sized keys or values being allocated in a tight loop. Results in Appendix L shows that Prefuzz is better than Murmur, and the large gains made by the Constant hash persist, but with less structure and more noise in the key sets, Prefuzz is becoming harder to beat.
- (`CONS :RND 6`) Similar to `FIXNUM :RND 6`, but we allocate real `cons` objects. See Appendix M for the results.
- (`SYMBOL :EXISTING`) Finally, we explore the case where keys are not allocated a tight loop that populates the hash table by using the set of existing `symbols` from Lisp as keys. Appendix N shows that despite the scarcity of structure in the key distribution, Prefuzz maintains a small advantage over Murmur. As expected, Co+Pr and Adaptive follow in suite, most of their advantage being in the Constant hash phase.

In all results presented, the Co+Pr and Adaptive hash functions, which both start out with the Constant hash, gain a lot of performance on insertion but lose on lookups. This is still an overall win based just on the numbers presented here except in very lookup-heavy workloads. However, an even larger unquantified benefit is in

[8] A fixnum is a signed integer in Lisp that fits into a machine word. It's similar to an `int64` on x86-64.

the reduced memory usage and garbage collection times due to the Constant hash having a specialized single vector implementation.

In summary, a general-purpose hash such as Murmur is a safe but suboptimal choice for many common situations in `eq` hash tables. SBCL's own Prefuzz hash is hand-crafted for these common situations, on which it is difficult to beat. Still, because it is non-adaptive, it sacrifices performance in other cases and has terrible worst-case behaviour. Our adaptive approach combines the worst-case safety of Murmur with the common case performance of Prefuzz. The adaptive approach even manages to slightly outperform Prefuzz because having the reliable safety net of the fallback mechanism allows it to be more aggressive in catering to the common case.

4.11 Macrobenchmarks

Microbenchmarks are useful indicators of the highest achievable throughput, but their results do not necessarily carry over to more complex workloads, where factors such as code size and complexity gain importance. To investigate this issue, we conducted experiments where hash table operations constitute only a small fraction of the workload. In particular, we measured the times to

(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.

Appendix G has the detailed results; here, we only provide a summary. Among the three benchmarking suites, the first one is the heaviest on `eq` and `equal` hash table operations. With SBCL's statistical profiler, we estimated that about 1.7% of the total runtime was spent in small `eq` hash tables (that is, within Constant hash's range of 0–32 keys) and 1.3% in larger ones, while operations on `equal` hash tables took 1%. The relative speedup was 8% in the large `eq` case and 50% for `equal`. The gain in the small `eq` case is harder to pin down because Constant hash's significantly reduced garbage collection cost; we estimate it to lie in the 8%–14% range.

In the second suite, small/large `eq` and `equal` hash table operations constituted 0.55%, 0.25% and 0.3% of the baseline result. Relative gains were as previously except for the large `eq` case, possibly because our benchmarking methodology is not able to detect differences so small, or because the increased code size is not worth it in code paths so cold.

Finally, we timed SBCL tests on a per-file basis and found no major performance regressions. Overall, we observed a 0.7% gain due to adaptive `eq` hashing, with adaptive `equal` hashing contributing another 1.5%, which came almost exclusively from the two tests with longer list keys (see Section 3.2).

In summary, by surviving the difficult transition from hot-path microbenchmarking to the cooler workloads reported in this section, adaptive hashing emerges as a method of practical relevance.

5 RELATED WORKS

Our method takes inspiration from Perfect Hashing, which selects a hash function for a given set of keys (known as static hashing). As its name implies, Dynamic Perfect Hashing [3, 6] allows the set of keys to change but still guarantees worst-case constant lookup time. However, it requires more memory than plain hash tables, so it is not a drop-in replacement for them. Cuckoo hashing also has worst-case constant lookup time but with a lower memory footprint. In a sequential implementation, it requires 1.5 hash function evaluations and memory accesses on average per lookup, which is about what the uniform hash has at load factor 1 (see Proposition 6).

Probably Dance [16], Rabbit Hashing [17] use the maximum probe length to decide when to grow the hash table or reseed the hash function but, unlike our adaptive method, they do not change its functional form, nor do they select the hash function to fit a given set of keys (see Algorithm 3).

VIP hashing [12] moves the more frequently accessed keys earlier in the collision chains to reduce the average lookup cost. Since adapting to the key access distribution is performed online, this method also needs to take extreme care to minimize overhead. In contrast to our work, they perform online adaptation of the hash table internal storage layout but leave the hash function constant.

Hentschel et al. [8] propose a method to learn the hash function offline from samples of the key distribution by using the most informative parts of compound keys. Our case study on string keys in Section 3 can be seen as an online version of their method, with the expensive learning phase removed.

In the taxonomy of Chi and Zhu [4], adaptive hashing falls under data-dependent hashing although they assume that the training is performed offline. The adaptive hashing method can also be viewed as a more robust, key-aware version of user-defined hash functions, which are also adapted offline to a particular key distribution.

There is a history of handcrafted hash functions that perform well in common cases, but exhibit spectacular failures in others. As we have seen, Prefuzz in SBCL is one such example. Java used to hash only about 1/8 of the characters in long strings [9]. This hardcoded limit made hashing faster, but as it could lead to lots of collisions without no fallback mechanism to save it, from JDK 1.2 on, all characters are hashed.

6 CONCLUSION

We have laid out the case for adaptive hash functions, which reside between key-agnostic and perfect hashes. Our primary contribution lies in reconceptualizing hash tables as inherently adaptive data structures, which can marry the theoretical guarantees of universal hashing with the common-case performance of weak hash functions. To support this viewpoint, we implemented this approach in a real-life system and demonstrated improved performance as well as robustness on string and integer/pointer hashing by capturing real-life key patterns and providing efficient search algorithms. The design space opened up by the adaptive hashing framework is large, and the adaptation mechanisms investigated in this work hardly cover a substantial or particularly imaginative part of it, leaving ample room for further developments. In particular, the max-chain-length mechanism can form the basis of a defense against denial-of-service collision attacks without constraining the choice of hash function [2].

Finally, the source code of the SBCL changes and the benchmarking code to reproduce the experimental results are open-sourced and available at https://github.com/melisgl/sbcl/tree/adaptive-hash.

ACKNOWLEDGMENTS

We thank Christophe Rhodes, Miloš Stanojević, Andrew Senior, Paul-Virak Khuong, and the reviewers for their valuable comments.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

[2] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[3] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

[4] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)*, 50(1):1–36, 2017.

[5] William Collins. *Data Structures and the Java Collections Framework*. McGraw-Hill Science/Engineering/Math, 2004. ISBN 0073022659.

[6] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[7] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[8] Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-learned hashing: Constant time hashing with controllable uniformity. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1640–1654, 2022.

[9] JDK bug database. JDK-4045622: java.lang.String.hashCode spec incorrectly describes the hash algorithm. https://archive.fo/LB0wY, 1997. Accessed: 2024-04-14.

[10] JDK bug database. JDK-4669519: Hashmap.get() in JDK 1.4 performs very poorly for some hashcodes. https://bugs.java.com/bugdatabase/view_bug?bug_id=4669519, 2023. Accessed: 2024-04-14.

[11] Bob Jenkins. Integer hashing. https://web.archive.org/web/20070210182431/http://burtleburtle.net/bob/hash/integer.html, 2007.

[12] Aarati Kakaraparthy, Jignesh M Patel, Brian P Kroth, and Kwanghyun Park. VIP hashing – Adapting to skew in popularity of data on the fly (extended version). *arXiv preprint arXiv:2206.12380*, 2022.

[13] Hans Peter Luhn. A new method of recording and searching information. *American Documentation*, 4(1):14–16, 1953.

[14] Bernard ME Moret. Towards a discipline of experimental algorithmics. In *Data structures, near neighbor searches, and methodology*, pages 197–213. Citeseer, 1999.

[15] William Newman. SBCL: Steel Bank Common Lisp, 1999. URL https://sbcl.org.

[16] Probably Dance. I wrote the fastest hash table, 2017. URL https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/. Accessed: 2024-04-14.

[17] Rabbit Hashing. Rabbit hashing, 2015. URL https://github.com/tjizep/rabbit. Accessed: 2024-04-14.

[18] Kyle Siegrist. Probability, mathematical statistics, stochastic processes. https://www.randomservices.org/random/urn/Birthday.html, 1997.

[19] Guy Steele. *Common LISP: The language.* Elsevier, 1990.

[20] Wikipedia contributors. Hash table — Wikipedia, The Free Encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Hash_table&oldid=1207615479. [Online; accessed 3-March-2024].

[21] Wikipedia contributors. Fowler–Noll–Vo hash function — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Fowler%E2%80%93Noll%E2%80%93Vo_hash_function&oldid=1215467221, 2024. [Online; accessed 14-April-2024].

A PROOFS

We restate the propositions from Section 2 and Section 4.5 and provide proofs.

Proposition 6. *[Expected Cost of the Uniform Hash] Let P be a uniform distribution over functions that map keys to buckets. Then,*

$$\mathop{\mathbb{E}}_{\pi_{1:n}\sim P} C(c([\pi_1(k_1),\dots\pi_n(k_n)],m)) = 1+\frac{n-1}{2m},$$

where $\pi_{1:n}$ are n independent samples from P.

Proof. Because we sample a hash function independently for each key, $\pi_i(k_i)$ are independent and distributed uniformly over the key space. Writing the expected number of comparisons for a lookup as a sum over the ith key added to the hash table, we get

$$\frac{\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right)}{n} = \frac{n+\frac{n(n-1)}{2m}}{n} = 1+\frac{n-1}{2m}. \qquad \square$$

Proposition 4. *[Perfect Hashes have Minimal Cost] Let $U(n,m)$ be the bucket count vector of any perfect hash of n keys and m buckets. Let $q=\lfloor n/m\rfloor$ and $r = n \bmod m$. Then,*

$$C(U(n,m)) = (m-r)\frac{q(q+1)}{2n} + r\frac{(q+1)(q+2)}{2n},$$

and this cost is minimal.

Proof. A set of hash values is either perfect or non-perfect. Since all perfect hashes have the same cost $C(U(n,m))$, if we could construct a perfect hash with lower cost from any non-perfect hash, it would follow that $C(U(n,m))$ is minimal.

Next, we show one such construction. For any non-perfect set of hash values with bucket counts c, there are always two buckets i and j such that $c_i > c_j + 1$ (else it would be a perfect hash due to bucket counts having to sum to n). By moving one hash from bucket i to j, we get a new set of hash values with bucket counts c' whose cost is lower because $2n(C(c)-C(c')) = c_i(c_i+1)+c_j(c_j+1)-(c_i-1)c_i-(c_j+1)(c_j+2) = c_i - c_j - 1 > 0$. $\square$

Proposition 7. *[Expected Regret of the Uniform Hash] For all load factors $q \in \mathbb{N}$ ($n = qm$), the expected regret of the uniform hash is $0.5+\frac{1}{m}$.*

Proof. Let $\pi_{1:qm}(k_{1:qm}) = [\pi_1(k_1),\dots,\pi_{qm}(k_{qm})]$. Then,

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\pi_{1:qm}} R(\pi_{1:qm}(k_{1:qm}),m)\\
&\quad= \mathop{\mathbb{E}}_{\pi_{1:qm}} C\big(c(\pi_{1:qm}(k_{1:qm}),m)\big) - C\big(U(n,m)\big)\\
&\quad= 1+\frac{qm-1}{2m}-\frac{q+1}{2}\\
&\quad= 0.5+\frac{1}{m}.
\end{aligned}$$ $\square$

Proposition 8. *[Expected Cost of Pointer-Mix] Let $k_{1:n}$ be integer keys, and $\mathcal{P} = \{k_i \gg \mathrm{PB} : i \in [1,n]\}$ the set of pages (the high bits of keys). Let the keys be distributed over the pages uniformly, $n = |\mathcal{P}|u$, where u is the number of keys on the same page ($u = |\{i\colon k_i \gg \mathrm{PB} = p\}|$ for all pages $p \in \mathcal{P}$). We assume that all u keys on the same page form random subsets of arithmetic progressions with page specific offsets but the same increments. Then, the expected cost of the Pointer-Mix hash function is*

$$1+\frac{n-u\min(1,\frac{2^{\mathrm{PB}-s}}{m})}{2m}.$$

Proof. First, we look at the case where there can be no collisions between keys on the same page. This is true if the keys form an arithmetic progression and are not just random subsets. Due to the subset assumption, it is also true if the hash table is large enough to hold a page worth of keys (shifted by s): $log_2(m) \geqslant \mathrm{PB} - s$.

The cost decomposes as the sum of the number of hashes in the same bucket as keys are added one by one. Thus, when there are $(p-1)$ previous pages' worth of keys already in the hash table, all u keys on the next page will contribute the same amount $1+(p-1)\frac{u}{m}$ to the cost because there are no collisions between them.

$$C = n^{-1}\sum_{p=1}^{|\mathcal{P}|} u\left(1+(p-1)\frac{u}{m}\right) = 1+\frac{n-u}{2m}.$$

Second, if keys (shifted by s) on the same page may collide randomly modulo m, then we can expect to get only $um^{-1}2^{\mathrm{PB}-s}$ guaranteed no colliding keys. Updating our formula, we get that

$$C = 1 + \frac{n - u\min(1, \frac{2^{\mathrm{PB}-s}}{m})}{2m}. \qquad \square$$

B THE EXPECTED NUMBER OF COLLISIONS

Here, we derive computationally cheap upper bounds on the expected number of collisions with the uniform hash for testing whether the observed number of collisions in Algorithm 1 is too many.

Given n keys, m buckets, and hash values $h_{1:n}$, let c and u be the number of collisions and unused (empty) buckets, respectively. The number of used buckets $m - u$ is equal to the number of non-colliding keys $n - c$, so $c = u + n - m$ and it suffices to bound u to bound c. Appealing to the birthday problem [18], the expected number of unused buckets is

$$\mathbb{E}\, u = m\left(1 - \frac{1}{m}\right)^n.$$

Holding the load factor $f = n/m$ constant, the proportion of unused buckets is monotonically increasing in m and

$$\lim_{m\to\infty} \mathbb{E}\,\frac{u}{m} = \lim_{m\to\infty} \mathbb{E}\left(\left(1 - \frac{1}{m}\right)^m\right)^{n/m} = \exp\left(-\frac{n}{m}\right) = \exp(-f),$$

using the product limit formula of the exponential function.

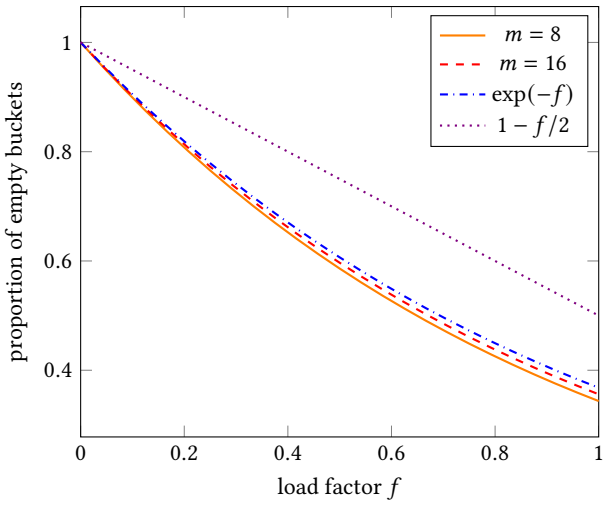

Figure 9: Expected proportion of empty buckets with the uniform hash given a load factor with the two smallest possible hash table sizes ($m = 8$ and $m = 16$), the tight upper bound $\exp(-f)$, and the loose but cheap upper bound $1 - f/2$ used for small sizes.

Based on this upper bound, for small eq and all equal hash tables, we use the test $m\exp(-f) < 0.9u$ (where $u = m + c - n$) to indicate whether the expected number of unused buckets (hence collisions) is too high.

For eq hash tables when $m < 2048$, we use a faster to compute and looser upper bound. In SBCL's hash table implementation, the load factor cannot exceed 1, so we only need to consider the $[0, 1]$ interval. In this interval, $\exp(-f) \leqslant 1 - f/2$. Then, from $\mathbb{E}\,\frac{u}{m} \leqslant \exp(-f)$, we have that

$$\begin{aligned} \mathbb{E}\,\frac{u}{m} &\leqslant 1 - f/2 \\ \mathbb{E}\, u &\leqslant m - n/2 \\ \mathbb{E}\, c + m - n &\leqslant m - n/2 \\ \mathbb{E}\, c &\leqslant n/2. \end{aligned}$$

Since this upper bound is already quite loose at most load factors, we use $c > (n \gg 1)$ to test for too many collisions at small hash table sizes.

C SBCL HASH TABLES

SBCL hash tables are technically separate-chaining, meaning that there are explicit chains of keys which fall into the same bucket. Ironically for a Lisp, these chains are not lists: for performance reasons, they are represented by two arrays of indices, called the `index-vector` and the `next-vector`), which are only resized when the hash table grows. The main pieces fit together as follows:

- The vector `pairs` holds alternating keys and values in a stable order, which is important for iteration (e.g. `maphash`). The first key is at index 2.
- `index-vector` is a power-of-2-sized array of indices, that maps a bucket to the index of the first key–value pair in `pairs` or zero if it's empty.
- `next-vector` maps the index of a pair to the index of the next pair in the collision chain or zero at chain end. It also chains empty slots in `pairs` together.
- For all but the lightest hash functions (standard eq and eql tables), the hash values of all keys in the hash table are cached in `hash-vector`. At lookup, the cached hash is compared to the hash of the key being looked up, and if they are different, then we know without invoking the potentially expensive comparison function that they cannot match (Algorithm 1).

An important operation is *rehashing*. When the number of buckets increases, keys are reassigned ("rehashed") to buckets based on their hash values, which are taken from `hash-vector` if there is one. Rehashing iterates over `pairs`, rewriting `index-vector` and `next-vector`.

Each standard hash table type (`eq`, `eql`, `equal`, `equalp`) have separate accessors (GET, PUT, DEL), which are invoked through an indirect call, and the two lighter ones have the hash function and comparison function inlined. There is no SIMD, and SBCL does not devirtualize calls.

D IMPLEMENTATION DETAILS

`Eq` hash tables are initialized with the Constant hash and a `pairs` vector is allocated for up to 8 key–value pairs. At this time, there is no `index-vector` and `next-vector`. The `pairs` vector is doubled in size as more keys are added, but no rehashing is necessary as there are no chains yet. Once the number of keys exceeds 32, *count_common_prefix_bits* (see Algorithm 3) is invoked with 16 keys to guess the shift s, we switch to the normal SBCL hash table

implementation, set the hash function to *pointer_shift* and rehash (Algorithm 4). Switching to the new hash function is implemented as changing the set of accessors.

We added new hash table accessors for Murmur and added a new slot for the detected shift s to the hash table structure. Instead of adding separate accessors for Pointer-Shift, which actually had the best performance in microbenchmarks, we made Pointer-Shift and Prefuzz share accessors, and to the inlined hash function we added an $if\ s = 0$ that dispatches to Prefuzz. This is to reduce pressure on the instruction cache, which is an important consideration in macrobenchmarks. To minimize the performance penalty on rehashing, we lift this dispatch out of the rehashing loop in the same way that dispatches to different accessors are.

For `equal` hash tables, we pack the truncation limit and the current max-chain-length into a single machine integer and store it in the same slot that we used for s in the `eq` case.

E BENCHMARKING ENVIRONMENT

All reported results are from a single Intel Core i7-1185G7 laptop running Linux with the `performance` scaling governor. Turbo boost and CPU idle states were disabled.

```
echo performance > /sys/firmware/acpi/platform_profile
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
cpupower -c 3 idle-set -D 0
```

The benchmarking process was run at maximum priority (`nice -n -20`), pinned to a single CPU (`taskset -c 3`).

F MICROBENCHMARKING METHODOLOGY

We plot the estimated time it takes to do a particular operation vs the number of keys (between 1 and at most 2^{24}) in the hash table for different key types and allocation patterns. All hash table variants compared in this paper allocate memory only when the insertion of a new key requires growing the internal data structures.[9] To reduce the computational burden of the benchmarking process, we list ranges of key counts within which no resizing takes place and only measure performance at the minimum and maximum key count in each range, assuming that linear interpolation is a reasonable approximation in between.

At each such key count, we then estimate the average time it takes to perform a hash table operation (e.g. PUT). First, a set of keys is generated for the given type (e.g. `FIXNUM`) and allocation pattern (e.g. `:RND 6`). The hash table is allocated in an empty state with comparison function `equal` for string keys or `eq` in all other cases. Then, we measure the average time it takes to perform a given operation for keys in the key set:

- PUT: Inserting one key when populating the hash table with all keys in the key set. Keys are inserted in the order they were generated.
- GET: Looking up a key in the key set. Keys are looked up in random order.
- MISS: Looking up a key not in the key set. Keys are looked up in random order.

[9] Moreover, they do so in an identical and deterministic pattern, so we exclude the time spent in garbage collection from the measurements. The exception to this pattern is the Constant hash, whose specialized implementation has a smaller than usual memory usage.

Table 1: Estimated means and relative standard errors of real (wall clock) and CPU times in seconds to *compile and load* a set of libraries.

	Real time	±RSE%	CPU time	±RSE%
Pr	24.068	0.02%	24.037	0.02%
Mu	24.152	0.01%	24.117	0.01%
Co+Pr	23.976	0.03%	23.945	0.02%
Co+Mu	23.979	0.03%	23.943	0.03%
Co+Pr>Mu	23.988	0.02%	23.955	0.02%
Co+PS>Pr>Mu	23.951	0.02%	23.918	0.01%
Equal*	23.824	0.02%	23.792	0.02%

Table 2: Estimated times to *test* a set of libraries.

	Real time	±RSE%	CPU time	±RSE%
Pr	25.512	0.03%	24.493	0.02%
Mu	25.632	0.05%	24.632	0.04%
Co+Pr	25.367	0.03%	24.352	0.02%
Co+Mu	25.360	0.04%	24.342	0.03%
Co+Pr>Mu	25.385	0.04%	24.367	0.03%
Co+PS>Pr>Mu	25.372	0.03%	24.374	0.02%
Equal*	25.257	0.03%	24.256	0.02%

- DEL: Deleting a key in the key set. Keys are deleted in random order.

The above steps are performed in the order listed. That is, first the hash table is populated, and PUT is measured, followed by GET then MISS. Finally, DEL timings are taken. At the end of the DEL phase, the hash table is once again empty.

These average times over key sets have a low relative standard deviation of about 0%–2%. To reduce the variance further, we take multiple such measurements and report their average. In particular, we take at least 3 measurements, then continue until the total number operations performed exceeds 5 000 000.

Finally, when the number of keys is less than 100, timing granularity is a limiting factor, so we allocate a number of hash tables (plus the two key sets for each, the second being for MISS) and measure the total time it takes to e.g. populate them. The number of hash tables is chosen such that the total number of keys is at least 100. This is a low enough number that the total memory footprint of the allocated hash tables stays below 32KB, the CPU's L1 cache size in our benchmarking environment.

SBCL had a 16GB heap.

G MACROBENCHMARK RESULTS

Here, we describe our macrobenchmarking experiments in Section 4.11 in more detail. In the first suite, 16 libaries were compiled and loaded with `(asdf:load-system <library> :force t)`. In the second, the tests of the same libraries were run with `(asdf:test-system <library>)`. In the third, SBCL tests were run with file-by-file with `tests/run-tests.sh`.

We compared the following configurations (following a naming convention like in Algorithm 4):

Table 3: Estimated times to *run SBCL tests*.

	Real time	±RSE%	CPU time	±RSE%
Pr	582.164	0.02%	452.344	0.03%
Mu	582.858	0.02%	453.069	0.02%
Co+Pr	580.071	0.02%	450.294	0.03%
Co+Mu	579.448	0.02%	449.644	0.03%
Co+Pr>Mu	578.909	0.02%	449.035	0.02%
Co+PS>Pr>Mu	578.207	0.02%	448.392	0.03%
Equal*	568.890	0.02%	439.091	0.02%

(1) **Pr**: unchanged baseline version with the Prefuzz `eq` hash and the non-adaptive `equal` hash
(2) **Mu**: the `eq` hash was changed to Murmur3
(3) **Co+Pr**: Constant hash followed by Prefuzz
(4) **Co+Mu**: Constant hash followed by Murmur3
(5) **Co+Pr>Mu**: Constant hash followed by Prefuzz with fallback to Murmur3
(6) **Co+PS>Pr>Mu**: Constant hash followed by Pointer-Shift with fallback to Prefuzz then to Murmur3 (called Adaptive in Section 4.10).
(7) **Equal***: Like the previous, but the `equal` hash is also adaptive (see Section 3).

Note that since Pr and Co+Pr lack an eventual fallback to Murmur, they have an easy-to-trigger unbounded worst case, so it may be more pertinent to think of Mu as the baseline.

In the benchmarking environment described in Appendix E, all individual benchmarks in the three benchmark suites (e.g. running the tests of a single library or a single SBCL test file) were run 10 times on each configuration with their runs interleaved to reduce the effect of correlated noise. We estimated the mean time a benchmark took on each configuration and computed the standard error of this estimated mean. We report the total estimated mean times for each benchmark suite along with their relative standard errors (RSE) in Tables 1 to 3.

H RESULTS FOR STRINGS

Keys are sampled from the set of all (about 40 000) strings in the running Lisp. This includes names of symbols, packages, docstrings, etc. The 10th and 90th percentiles of the distribution of the string lengths is 7 and 39. The MISS key set consists of random strings with length sampled uniformly from the $[4, 44]$ interval.

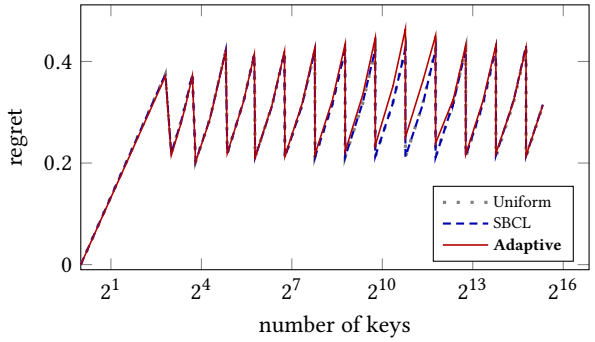

Figure 1: Regret (Definition 5) with string keys. Adaptive does not gain or significantly compromise on regret. Points where the truncation limit changes vary between runs.

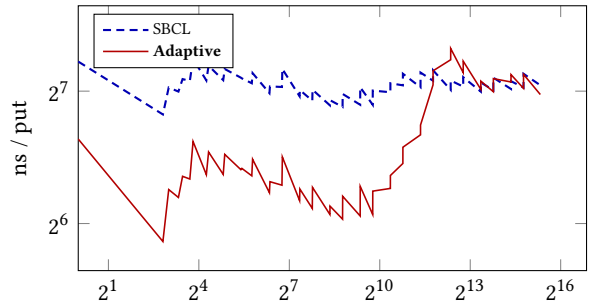

Figure 2: PUT timings in nanoseconds with string keys. Note the log scales. The plot shows the *average* time for inserting a new key when populating an empty hash table with a given number of keys.

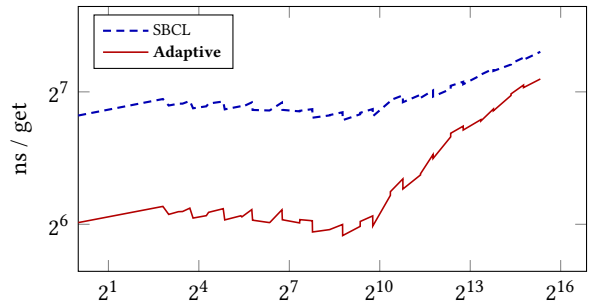

Figure 3: GET timings with string keys.

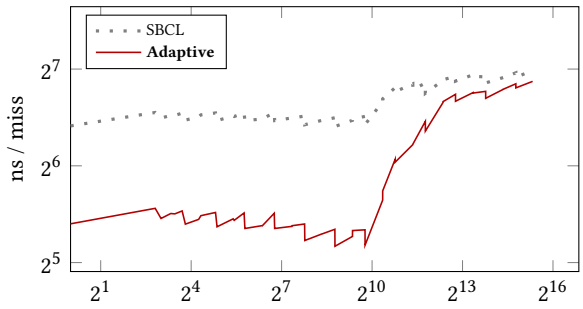

Figure 10: MISS timings with string keys

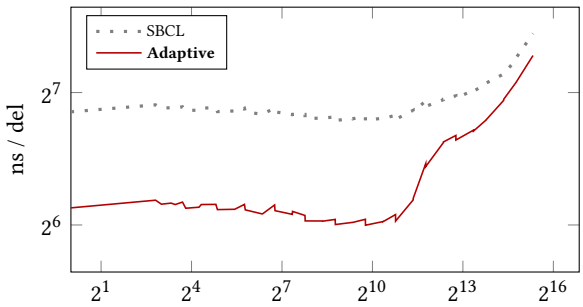

Figure 11: DEL timings with string keys

I RESULTS FOR `FIXNUM :PROG 1`

Keys form an arithmetic progression with difference 1 starting from a large random offset and are used in that order for PUT. We also generate a set of keys not in the hash table for MISS, by using another, suitable offset (so that the two sets are disjunct). For GET, MISS, and DEL, keys are presented in random order.

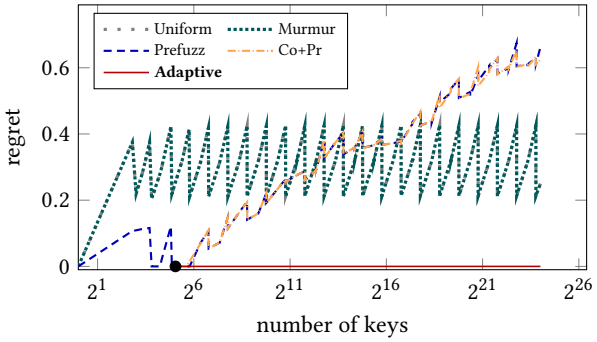

Figure 4: Regret with `FIXNUM :PROG 1`. Murmur closely tracks Uniform. Prefuzz is aggressively optimized for small sizes. Adaptive (Algorithm 4) is a perfect hash here. Both Co+Pr (Constant followed by Prefuzz) and Adaptive use the Constant hash until the fixed switch point at 32 keys (black dot).

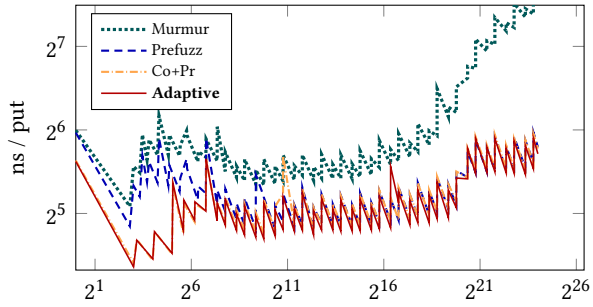

Figure 5: PUT timings with `FIXNUM :PROG 1`. Prefuzz outperforms Murmur even at large sizes despite higher regret because it's friendlier to the cache (its collisions are between subsequent elements of the progression), and its combination with Constant is even faster. Thus, despite being a perfect hash, Adaptive can improve on them only marginally.

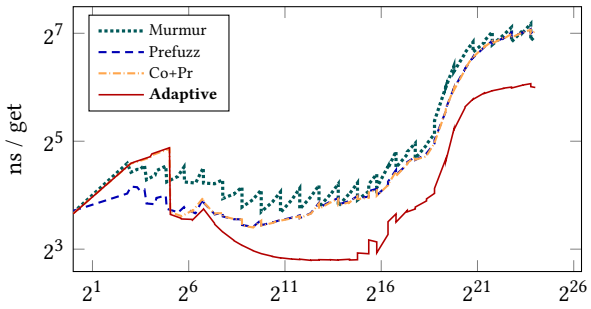

Figure 6: GET timings with `FIXNUM :PROG 1`. Keys are queried in random order so regret matters more here than with PUT, but the cache-friendliness of Prefuzz still keeps it ahead of Murmur. As expected, Adaptive can finally benefit from its zero regret after the Constant hash phase.

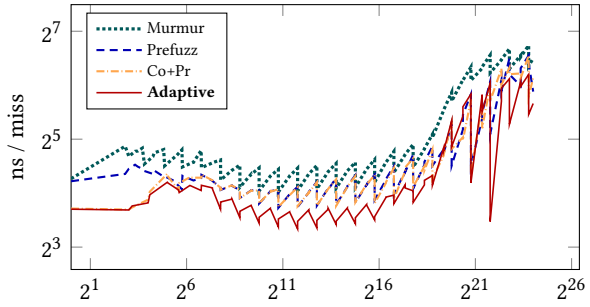

Figure 12: MISS timings with `FIXNUM :PROG 1`

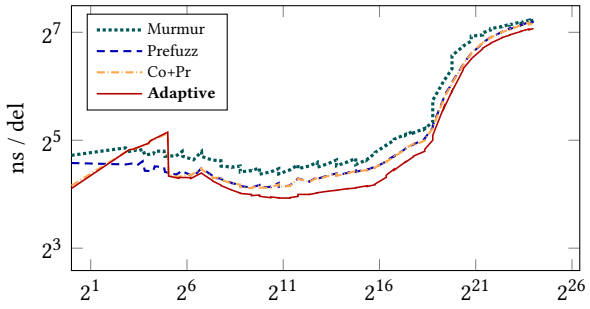

Figure 13: DEL timings with `FIXNUM :PROG 1`

J RESULTS FOR FLOAT :PROG 1

Keys are generated as in Appendix I, but the `fixnum` values are converted to `single-float`.

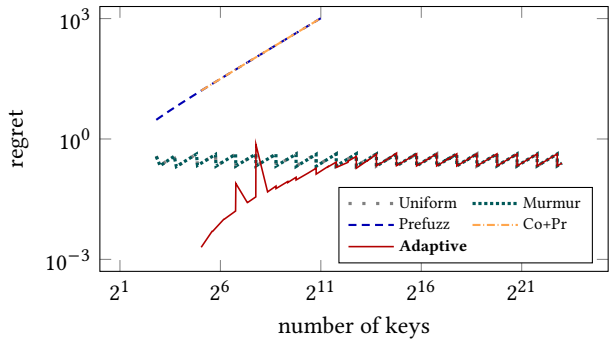

Figure 7: Regret with FLOAT :PROG 1. To be able to plot the catastrophic failure of Prefuzz (and of Co+Pr, consequently), we use log scale for regret on this graph. Single floats are especially problematic for Prefuzz because they can have many constant low bits. Adaptive detects these constant low bits and does better than Uniform until variation in the floating point exponents makes its original estimate of the number of constant low bits invalid, and the resulting gradual increase in collisions makes it switch to Prefuzz at rehash time. This is a spectacularly bad idea in this scenario, and the high number of collisions causes an immediate switch to Murmur. The switch times vary by hash table because the key sets are generated starting from random offsets.

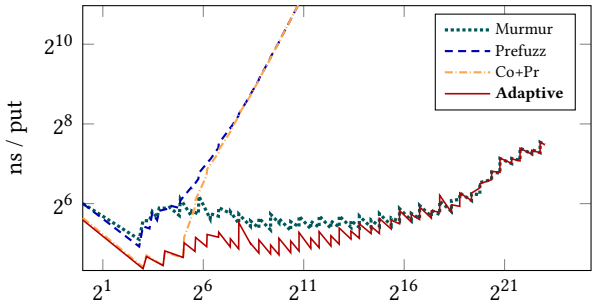

Figure 14: PUT timings with FLOAT :PROG 1

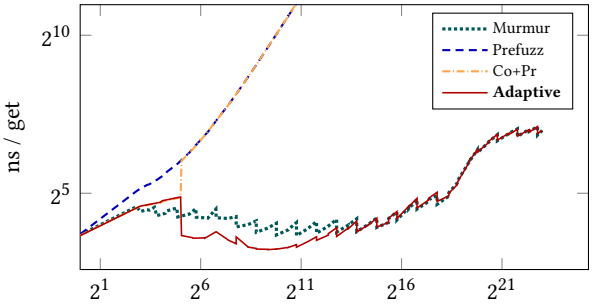

Figure 15: GET timings with FLOAT :PROG 1

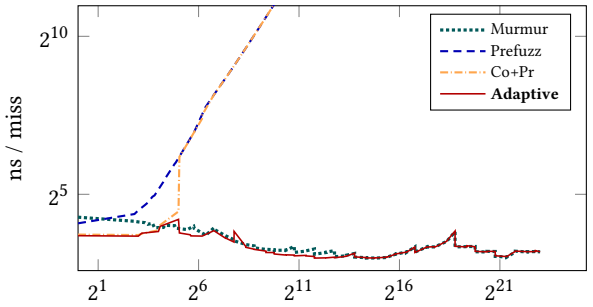

Figure 16: MISS timings with FLOAT :PROG 1

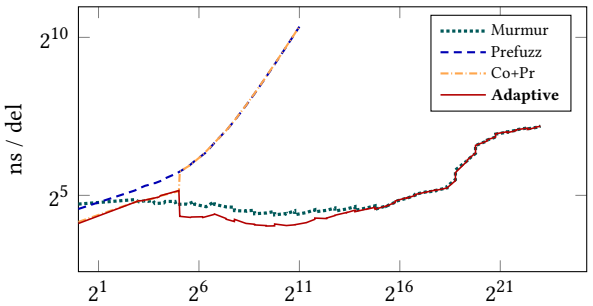

Figure 17: DEL timings with FLOAT :PROG 1

K RESULTS FOR `FIXNUM :PROG 12`

Same as `FIXNUM :PROG 1`, but with a difference of 12.

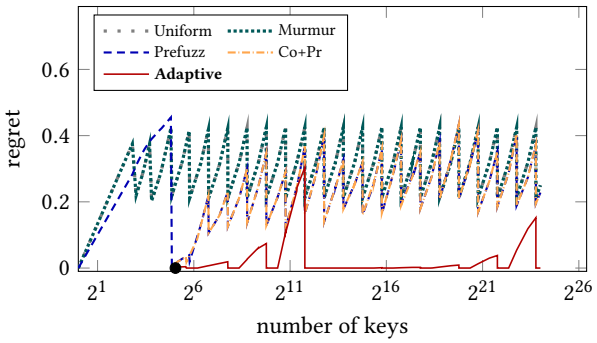

Figure 8: Regret with `FIXNUM :PROG 12`. Murmur closely tracks Uniform, but Prefuzz is better across almost the whole range. Arithmetic (Section 4.4) would be a perfect hash here, but Adaptive, which uses Pointer-Shift (Section 4.6), is not quite perfect due to the interference of its $k \gg \text{PB}$ term.

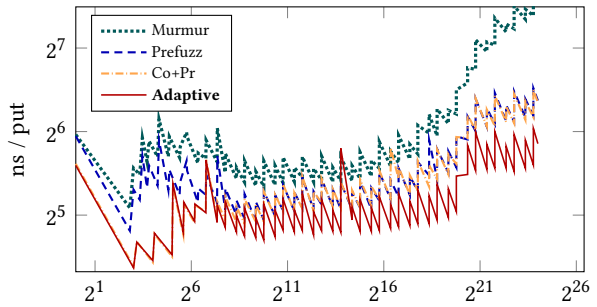

Figure 18: PUT timings with `FIXNUM :PROG 12`. Prefuzz outperforms Murmur due to its speed, lower regret and cache-friendliness. Adaptive is able to benefit from its advantage in regret only at larger sizes.

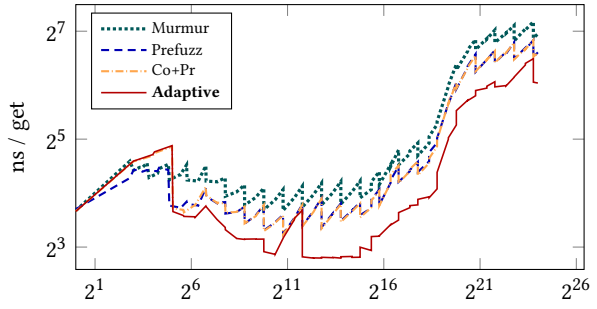

Figure 19: GET timings with `FIXNUM :PROG 12`. Compared to PUT, differences in regret translate more clearly to lookup performance because keys are queried in random order.

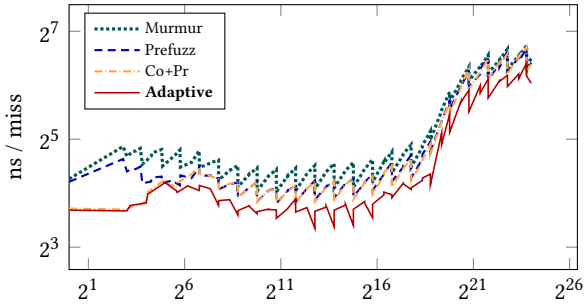

Figure 20: MISS timings with `FIXNUM :PROG 12`

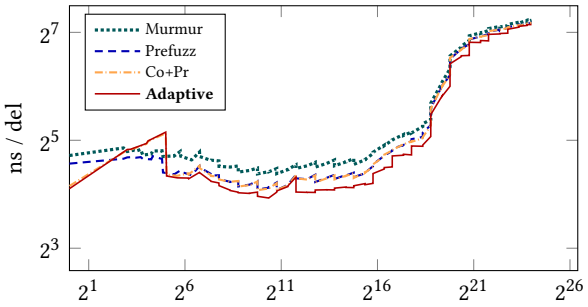

Figure 21: DEL timings with `FIXNUM :PROG 12`

L RESULTS FOR `FIXNUM :RND 6`

Similar to `:PROG 6`, but the difference between successive keys is sampled uniformly from the $[0, 5]$ interval.

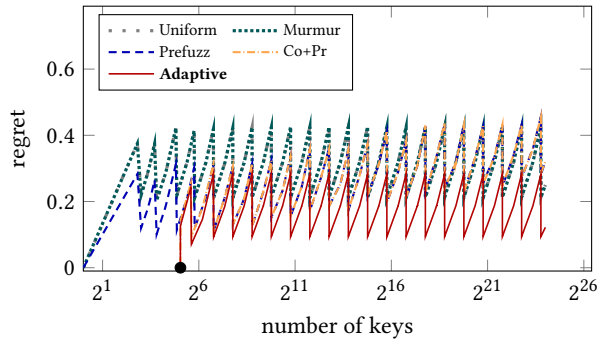

Figure 22: Regret with `FIXNUM :RND 6`. Prefuzz does better than Murmur initially but gradually loses its edge. Adaptive keeps its edge.

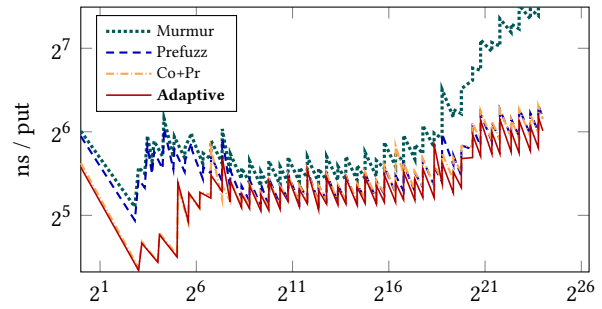

Figure 23: PUT timings with `FIXNUM :RND 6`. Once again, the advantage of Prefuzz over Murmur grows with size because it is friendlier to the cache. Adaptive manages to translate some of its lead in regret into outright speed.

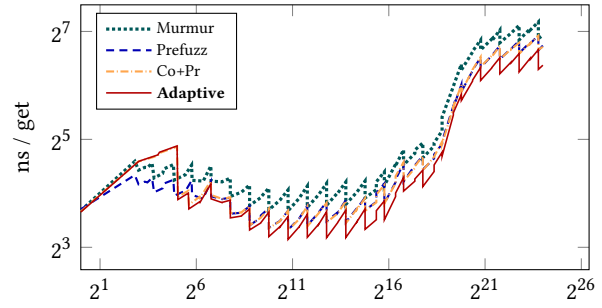

Figure 24: GET timings with `FIXNUM :RND 6`. Prefuzz is considerably ahead of Murmur at all sizes. Adaptive is better than Prefuzz at larger sizes.

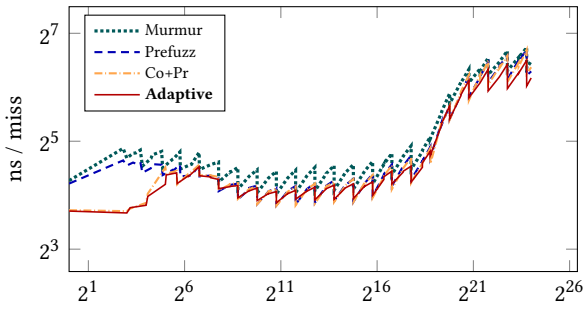

Figure 25: MISS timings with `FIXNUM :RND 6`

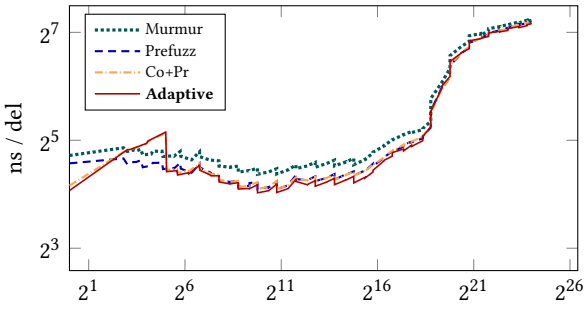

Figure 26: DEL timings with `FIXNUM :RND 6`

M RESULTS FOR CONS :RND 6

Like the FIXNUM :RND 6, but keys are cons objects with a random number of conses allocated between them. There is no explicit random offset is here; we rely on the addresses assigned by the memory allocator. To prevent the garbage collector from compacting memory regions holding these objects in memory, the skipped over conses are kept alive.

In all experiments, there is some unexplained weirdness at large sizes, which affects all hashes except Murmur. Even the regret of Adaptive is affected: it improves to an unexpected level but does so very erratically.

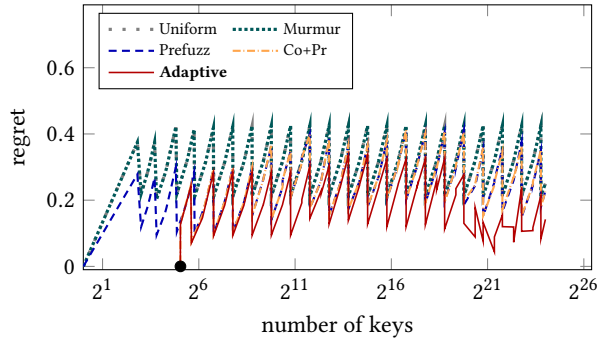

Figure 27: Regret with CONS :RND 6. Murmur closely tracks Uniform. Prefuzz works well at small sizes. Adaptive is a bit better still.

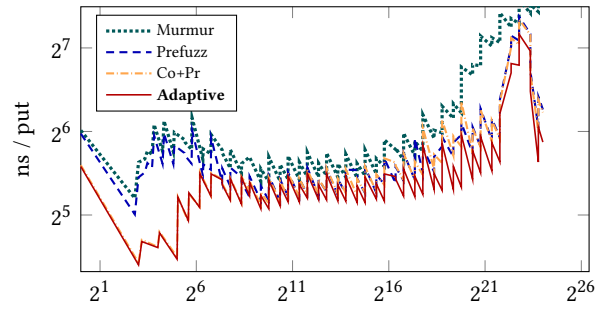

Figure 28: PUT timings with CONS :RND 6. Prefuzz outperforms Murmur even at large sizes despite its higher regret because its collisions are between subsequent elements of the progression, which is friendly to the cache.

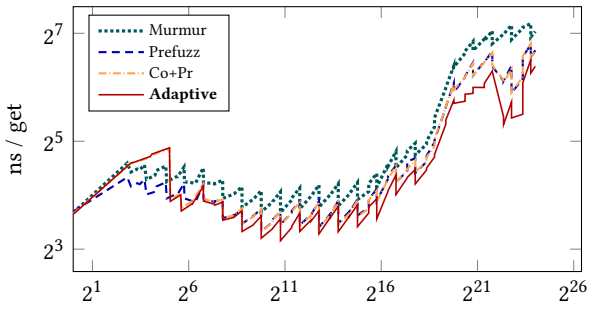

Figure 29: GET timings with CONS :RND 6

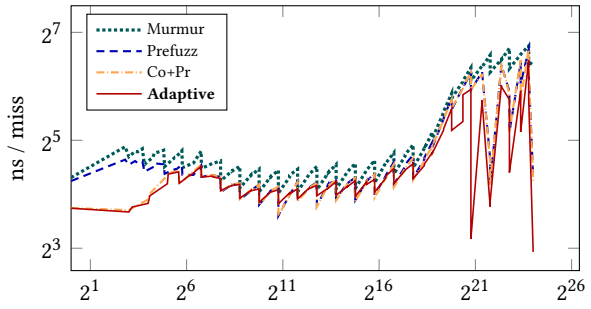

Figure 30: MISS timings with CONS :RND 6

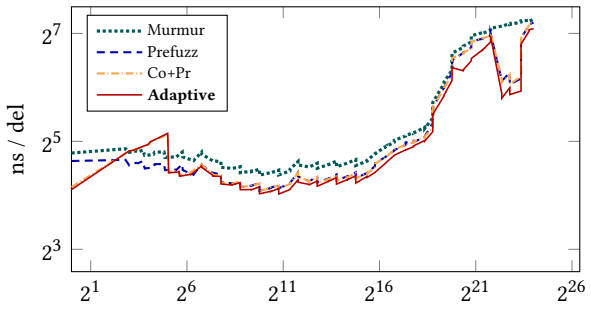

Figure 31: DEL timings with CONS :RND 6

N RESULTS FOR SYMBOL :EXISTING

We list all symbols in the running Lisp system and take a random subset of the desired size. These symbols happen to be packed tightly in not too many pages, and their addresses, if sorted, would approximately follow an arithmetic progression, which is a great fit for Pointer-Shift (Section 4.6, Algorithm 4). But the keys are randomly selected and presented in random order even for PUT, so this effect can only be seen at close to maximal sizes.

The MISS key set is generated with `(gensym)`.

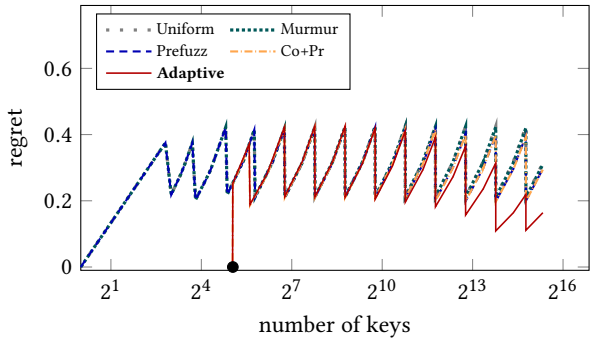

Figure 32: Regret with SYMBOL :EXISTING. All hashes closely track Uniform. Adaptive, which leaves memory addresses most intact, takes advantage of the nature of the data at close to maximal sizes.

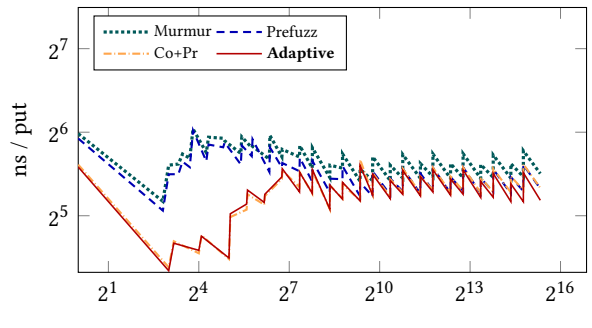

Figure 33: PUT timings with SYMBOL :EXISTING. Prefuzz outperforms Murmur because it is faster to compute. The advantage of Co+Pr and Adaptive over Prefuzz comes almost exclusively from the initial Constant hash phase, with Adaptive enjoying a small edge due to its better regret at the very end.

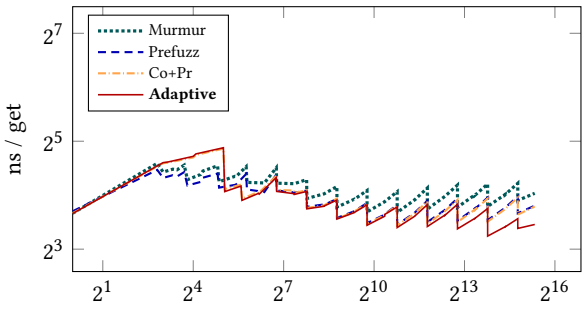

Figure 34: GET timings with SYMBOL :EXISTING.

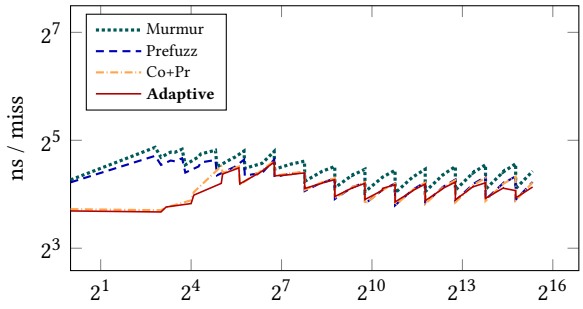

Figure 35: MISS timings with SYMBOL :EXISTING

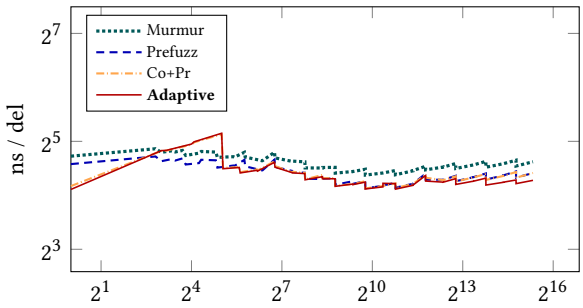

Figure 36: DEL timings with SYMBOL :EXISTING