Received: 11/22/2018 10:06 PM

Status: Closed
Priority: Normal
Assignee:
Target version:

Description

Background

It's common for applications to use string literals as hash keys, especially for applications that work with YAML or JSON:

    payload['name']

At the moment (r65895), hash lookups using a string key is about 30% slower than using a symbol key.

Proposal

We memoize the hash code for short fstrings. There is extra, currently unused space at the end of the RString struct we can use to store the hash code. The unique nature of fstrings makes it so that every user of each fstring benefit from the memoized hash.

Evaluation

The included benchmark hash_aref_fstr.rb is about 20% faster with this optimization. hash_aref_long_str.rb shows that for long strings which we cannot memoize, there is no visible performance penalty.

vm2_freezestring.yml is also not visibly slower after this optimization.

I have also attached a bechmark (string_key_vs_symbol_key.rb) that compares hash lookups with symbols against hash lookups with strings. With this optimization, the gap in performance is smaller. (10% slower post patch vs 30% slower on trunk)
Yes! I was one byte too conservative.

And wide-char encodings would need consideration.

I don't see why we need to special-case any encoding. RSTRING_EMBED_LEN gets the size in bytes and is used in String#bytesize. Do we write/read past arr[RSTRING_EMBED_LEN(str)] or modify frozen strings in-place somewhere?

I attached an updated patch that has one additional benchmark that tests the code path which writes the memo. No visible penalty there either.

#4 - 11/23/2018 07:24 PM - alanwu (Alan Wu)
- File 0001-Hash-code-memoization-for-short-fstrings-v3.patch added

#5 - 11/23/2018 11:01 PM - alanwu (Alan Wu)
- Description updated
- File 0001-Hash-code-memoization-for-short-fstrings-v4.patch added

#6 - 11/24/2018 05:35 AM - alanwu (Alan Wu)
Ah I see the problem with wide-char encodings now. There are some C APIs that write a codepoint-wide terminator at arr[RSTRING_EMBED_LEN], which can clobber the hash memo.
I will benchmark to see if fetching the encoding struct is slow; I think it's a cache miss every time.

#7 - 11/25/2018 03:22 AM - nobu (Nobuyoshi Nakada)
One idea is to shrink MEMO_HASH_LEN_MAX for a wide-char terminators.

#8 - 11/26/2018 04:39 AM - alanwu (Alan Wu)
- File 0001-Hash-code-memoization-for-short-fstrings-v5.patch added
I've taken multi-byte terminators into account and updated my implementation. I've also included a test that should warn us about hash clobbering.
I also noticed that the benchmark that creates many unique fstrings is actually 5% slower on v4 of this patch with a higher iteration count.
I had to change the hash table interaction for fstring creation a bit to make up the performance difference.
Apologies for the bigger patch size.

For short enough fstrings, we have extra space inside the RString struct we could use to remember the hash code.

Applications that work with JSON or YAML benefit from this optimization as they often use string literals as hash keys. The popular ORM ActiveRecord also uses fstring internally and thus would benefit from this optimization.

This also provides yet another incentive for users to turn on frozen string literals.

On a 64-bit machine, for strings that use ASCII and UTF-8 encoding, "short enough" means len < 16. For 32 bit machines, "short enough" means len < 8.

* st.c: introduce a new function: st_update_with_hash() that allows users to avoid unncessary re-hashing when doing multiple updates.

* string.c: memoize hash code for short, embeded fstrings after the embedded buffer. Make use of st_update_with_hash() to avoid hashing the same string multiple times.

* string.c: harden str_fill_term() to make sure we don't zero more than 4 bytes of memory. No encoding in the world needs a terminator longer than 4 bytes at the moment.

* hash_aref_fstr.rb: benchmark demonstrating the best case scenario. About 20% faster with this commit.

* hash_aref_long_str.rb: benchamrk demonstrating the worst case scenario for this optimization. It shows that there is no visible performance impact for strings that don't benefit from the optimization.

* freeze_unique_strings.yml: benchmark that creates many unique
fstrings. We memoize the hash for each fstring we create, but there is no visible performance impact.

I've taken multi-byte terminators into account and updated my implementation. I've also included a test that should warn us about hash clobbering.

Seems nice.

About the macro BUILDIN_SINGLE_BYTE_ENC_IDX_P, its name is misleading as it is true for UTF-8. In common, single-byte encoding means that it uses just one byte per character, 0..255 range, e.g., ASCII-8BIT, US-ASCII, and ISO-8859 family. UTF-8 is one of multi-byte encodings. Also it doesn't cover all non-wchar encodings, so it doesn't seem generic enough to be shared in encindex.h.

I had to change the hash table interaction for fstring creation a bit to make up the performance difference. Apologies for the bigger patch size.

It may be nice to separate st_update_with_hash patch.

I've taken multi-byte terminators into account and updated my implementation. I've also included a test that should warn us about hash clobbering.

Seems nice.

About the macro BUILDIN_SINGLE_BYTE_ENC_IDX_P, its name is misleading as it is true for UTF-8. In common, single-byte encoding means that it uses just one byte per character, 0..255 range, e.g., ASCII-8BIT, US-ASCII, and ISO-8859 family. UTF-8 is one of multi-byte encodings. Also it doesn't cover all non-wchar encodings, so it doesn't seem generic enough to be shared in encindex.h.

I had to change the hash table interaction for fstring creation a bit to make up the performance difference. Apologies for the bigger patch size.

It may be nice to separate st_update_with_hash patch.

nобu (Nobuyoshi Nakada) wrote:

About the macro BUILDIN_SINGLE_BYTE_ENC_IDX_P, its name is misleading as it is true for UTF-8.

I agree. I removed it.

It may be nice to separate st_update_with_hash patch.

Done.

I moved away from using +1 in the buffer size, since it could be +1 or +4 depending on the encoding. I also expanded the test to test a variety of string lengths.

Is there anything I can do to move this forward?

I proposed this change because every attribute access in Active Record goes through a hash lookup. When we profile our app in production, Active Record attribute access shows up as a clear hot spot.

When I used this patch to run response time benchmark against https://github.com/codetriage/codetriage, there was a 1% improvement. A 1% improvement would be translate to a great deal of cost savings for us in production.

I'm disappointed that the patch doesn't provide enough speed-up to justify the maintenance burden, but I thank you both for your time.

Isn't 1% a measurement error?

Does it always occur constantly?
nobu (Nobuyoshi Nakada) wrote:

Isn't 1% a measurement error?
Does it always occur constantly?

If a 1% improvement is consistently reproducible, then it's worth reconsidering.

alanwu (Alan Wu) wrote:

I proposed this change because every attribute access in Active Record goes through a hash lookup.
When we profile our app in production, Active Record attribute access shows up as a clear hot spot.

That doesn't come as a surprise to me.

When I used this patch to run response time benchmark against https://github.com/codetriage/codetriage, there was a 1% improvement. A 1% improvement would be translate to a great deal of cost savings for us in production.

Can you give more information? What 'response time benchmark' did you use (I didn't find any such benchmark in codetriage, but I didn't look very hard and am not familiar with that application)? Did you try other benchmarks, or can you try them?

I'm disappointed that the patch doesn't provide enough speed-up to justify the maintenance burden, but I thank you both for your time.

I don't think we say that 1% isn't enough. If it's reproducible, it's very easily worth it. But if we can't reproduce the speedup, it's difficult for us to include your code.

alanwu (Alan Wu) wrote:

After running some benchmarks on rubygems.org with wrk, I do not think this is patch is a good idea anymore. It seems to decrease throughput instead of increase it, despite the theoretical improvement shown in micro benchmarks.
The 1% improvement is there, but it's due to a change I made to ActiveRecord to make it benefit from hash memoization, not due to the hash memorization itself.
The better place to do this optimization is probably at compile time, after doing inlining, I think.
Thank you again for taking a look at this patch. I learned a lot trying to put it together.

KO1 (KOichi Sasada) wrote:

My opinion: For short strings, I don't think it doesn't improve. For long strings, it can be (not sure how many long string's hash value are used). So allocating hash value (+8 bytes) in external allocated area can help (but it should be applied to enough long strings).