**Ruby master - Feature #12142**

**Hash tables with open addressing**

03/04/2016 03:09 AM - vmakarov (Vladimir Makarov)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Hello, the following patch contains a new implementation of hash tables (major files st.c and include/ruby/st.h).

Modern processors have several levels of cache. Usually, the CPU reads one or a few lines of the cache from memory (or another level of cache). So CPU is much faster at reading data stored close to each other. The current implementation of Ruby hash tables does not fit well to modern processor cache organization, which requires better data locality for faster program speed.

The new hash table implementation achieves a better data locality mainly by:

- switching to open addressing hash tables for access by keys. Removing hash collision lists lets us avoid "pointer chasing", a common problem that produces bad data locality. I see a tendency to move from chaining hash tables to open addressing hash tables due to their better fit to modern CPU memory organizations. CPython recently made such switch (https://hg.python.org/cpython/file/ff1938d12240/Objects/dictobject.c).

PHP did this a bit earlier https://nikic.github.io/2014/12/22/PHPs-new-hashtable-implementation.html.

GCC has widely-used such hash tables (https://gcc.gnu.org/svn/gcc/trunk/libiberty/hashtab.c) internally for more than 15 years.

- removing doubly linked lists and putting the elements into an array for accessing to elements by their inclusion order. That also removes pointer chasing on the doubly linked lists used for traversing elements by their inclusion order.

A more detailed description of the proposed implementation can be found in the top comment of the file st.c.

The new implementation was benchmarked on 21 MRI hash table benchmarks for two most widely used targets x86-64 (Intel 4.2GHz i7-4790K) and ARM (Exynos 5410 - 1.6GHz Cortex-A15):

```
make benchmark-each ITEM=bm_hash OPTS='--r 3 -v' COMPARE_RUBY='<trunk ruby>'
```

Here the results for x86-64:

- `hash_aref_dsyr` 1.094
- `hash_aref_dsym_long` 1.383
- `hash_aref_fix` 1.048
- `hash_aref_flo` 1.860
- `hash_aref_miss` 1.107
- `hash_aref_str` 1.107
- `hash_aref_sym` 1.191
- `hash_aref_sym_long` 1.113
- `hash_flatten` 1.258
- `hash_ident_flo` 1.627
- `hash_ident_num` 1.045
- `hash_ident_obj` 1.143

05/08/2021 1/73
hash_ident_str 1.127
hash_ident_sym 1.152
hash_keys 2.714
hash_shift 2.209
hash_shift_u16 1.442
hash_shift_u24 1.413
hash_shift_u32 1.396
hash_to_proc 2.831
hash_values 2.701

The average performance improvement is more 50%. ARM results are analogous -- no any benchmark performance degradation and about the same average improvement.

The patch can be seen as

https://github.com/vnmakarov/ruby/compare/trunk...hash_tables_with_open_addressing.patch

or in a less convenient way as pull request changes

https://github.com/ruby/ruby/pull/1264/files

This is my first patch for MRI and may be my proposal and implementation have pitfalls. But I am keen to learn and work on inclusion of this code into MRI.

Related issues:
Related to Ruby master - Feature #5903: Optimize st_table (take 2) Closed
Related to Ruby master - Bug #13002: Hash calculations no longer using univer... Closed

Associated revisions
Revision 7575157 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)
Introduce table improvement by Vladimir Makarov vmakarov@redhat.com
[Feature #12142]
See header of st.c for improvement details.

You can see all of code history here:
<https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing>

This improvement is discussed at
<https://bugs.ruby-lang.org/issues/12142>
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.
* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.
* ext/-test-/st/foreach/foreach.c: catch up this change.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@56650 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 56650 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)
Introduce table improvement by Vladimir Makarov vmakarov@redhat.com
[Feature #12142]
See header of st.c for improvement details.

You can see all of code history here:
<https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing>

This improvement is discussed at
<https://bugs.ruby-lang.org/issues/12142>
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.
* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.

* ext/-test-/st/foreach/foreach.c: catch up this change.

Revision 56650 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)
Introduce table improvement by Vladimir Makarov vmakarov@redhat.com.
[Feature #12142]
See header of st.c for improvement details.

You can see all of code history here:
<https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing>

This improvement is discussed at
<https://bugs.ruby-lang.org/issues/12142>
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.

* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.
* ext/-test-/st/foreach/foreach.c: catch up this change.

Revision 56650 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)
Introduce table improvement by Vladimir Makarov vmakarov@redhat.com.
[Feature #12142]
See header of st.c for improvement details.

You can see all of code history here:
<https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing>

This improvement is discussed at
<https://bugs.ruby-lang.org/issues/12142>
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.

* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.
* ext/-test-/st/foreach/foreach.c: catch up this change.

Revision 56650 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)
Introduce table improvement by Vladimir Makarov vmakarov@redhat.com.
[Feature #12142]
See header of st.c for improvement details.

You can see all of code history here:
<https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing>

This improvement is discussed at
<https://bugs.ruby-lang.org/issues/12142>
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.

* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.
* ext/-test-/st/foreach/foreach.c: catch up this change.

**History**

#1 - 03/04/2016 03:59 AM - ko1 (Koichi Sasada)
Thank you for your great contribution.

Do you compare memory usages?
There are good and bad points.

- **Good**
  - removing fwd/prev pointer for doubly linked list
  - removing per bucket allocation

- **Bad**
  - it requires more "entries" array size. current st doesn't allocate big entries array for chain hash.
  - (rare case) so many deletion can keep spaces (does it collected? i need to read code more)

I think goods overcomes bads.

We can generalize the last issue as "compaction". This is what I didn't touch this issue yet (maybe not a big problem).

Trivial comments

- at first, you (or we) should introduce st_num_entries() (or something good name) to wrap to access num_entries/num_elements before your patch.
- I'm not sure we should continue to use the name st. at least, st.c can be renamed.

---

**#2 - 03/04/2016 05:50 AM - nobu (Nobuyoshi Nakada)**

Koichi Sasada wrote:

- at first, you (or we) should introduce st_num_entries() (or something good name) to wrap to access num_entries/num_elements before your patch.

It seems unnecessary for me to rename num_entries.

**#3 - 03/04/2016 08:54 AM - funny_falcon (Yura Sokolov)**

Great!

Notes:

- num_entries should remain num_entries. It is easier for you to change naming than fix all rubygems.
- do not change formatting of a code you do not change, it is awful to read and check that part of your patch.
- speed improvement is not from open addressing but from array storage for elements. You can use chaining with "next_index" with same result. But perhaps open addressing is simpler to implement.
- if you stick with open-addressing, then it could be even faster, if you store hashsum in st_entry.
- st_foreach is broken in case table were rebuilt.

Anyway, it is a great attempt I wished to do by myself, but didn't give time for. I hope, you'll fix all issues and change will be accepted.

**#4 - 03/04/2016 09:00 AM - funny_falcon (Yura Sokolov)**

Small note: there is no need in perturbation and 'python like' secondary hash cause ruby uses strong hash function. Better stick with quadrating probing.

**#5 - 03/04/2016 10:41 AM - mame (Yusuke Endoh)**

Great, but it seems to need more work. It consumes nearly 2 times much memory.

```
# trunk
$ /usr/bin/time -f %Mkb ./miniruby -e '([0..1000000].map { |i| i=>1 })'
280108kb

# hash_tables_with_open_addressing branch
$ /usr/bin/time -f %Mkb ./miniruby -e '([0..1000000].map { |i| i=>1 })'
500264kb
```

--

Yusuke Endoh mame@ruby-lang.org

**#6 - 03/04/2016 04:31 PM - vmakarov (Vladimir Makarov)**

Koichi Sasada wrote:

Thank you for your great contribution.

Thanks you for your quick response. I am not a Rubyist but I like MRI code.

05/08/2021
Sorry, I did not. Although I evaluated it. On my evaluation, in the worst case scenario, memory usage will be about the same as for the current hash tables taking into account that the element size is now 1/2 of the old size and the element array minimal usage is 50%. This is because, when the hash table rebuilds, the element array size is doubled. Rebuilding means 100% of array element usage before it. If it is less, the new array size will be the same or will be smaller.

This evaluation excludes cases when the current hash table uses packed elements (up to 6). But I consider it is a pathological case. The proposed hash tables can use the same approach. It is even more natural because the packed elements of the current hash tables have exactly the same structure as for the proposed table elements.

So the packed element approach could be implemented too for the proposed implementation. It means avoiding creation of entries array for small size tables. I don't see it is necessary unless the hash tables will be again used for method tables where most of them are small. Hash tables will be faster than the used binary search. But it is not a critical code (at least for benchmarks in MRI) as we search method table once for a method and all other calls of the method skips this search. I am sure you know it much better.

Speaking of measurements. Could you recommend credible benchmarks for the measurements. I am in benchmarking business for a long time and I know benchmarking may be an evil. It is possible to create benchmarks which prove opposite things. In compiler field, we use SPEC2000/SPEC2006 which is a consensus of most parties involved in the compiler business. Do Ruby have something analogous?

There are good and bad points.

- **Good**
  - removing fwd/prev pointer for doubly linked list
  - removing per bucket allocation
- **Bad**
  - it requires more "entries" array size. current st doesn't allocate big entries array for chain hash.
  - (rare case) so many deletion can keep spaces (does it collected? i need to read code more)

In the proposed implementation, the table size can be decreased. So in some way it is collected.

Reading the responses to all of which I am going to answer, I see people are worrying about memory usage. Smaller memory usage is important for better code locality too (although a better code locality does not mean a faster code automatically -- the access pattern is important too). But I consider the speed is the first priority these days (especially when memory is cheap and it will be much cheaper with new coming memory technology).

In many cases speed is achieved by methods which requires more memory. For example, Intel compiler generates much bigger code than GCC to achieve better performance (this is most important competitive advantage for their compiler).

This is actually seventh variant of hash tables I tried in MRI. Only this variant achieved the best average improvement and no any benchmark with worse performance.

I think goods overcomes bads.

Thanks, I really appreciate your opinion. I'll work on the found issues. Although I am a bit busy right now with work on GCC6 release. I'll have more time to work on this in April.

We can generalize the last issue as "compaction".
This is what I didn't touch this issue yet (maybe not a big problem).

Trivial comments
• at first, you (or we) should introduce st_num_entries() (or something good name) to wrap to access num_entries/num_elements before your patch.
• I'm not sure we should continue to use the name st. at least, st.c can be renamed.

Ok. I'll think about the terminology. Yura Sokolov wrote that changing entries to elements can affect all rubygems. I did not know about that. I was reckless about using a terminology more familiar for me.

• I always confuse about "open addressing" == "closed hashing" https://en.wikipedia.org/wiki/Open_addressing

Yes, the term is confusing but it was used since 1957 according to Knuth.

#7 - 03/04/2016 04:35 PM - vmakarov (Vladimir Makarov)
Nobuyoshi Nakada wrote:

Koichi Sasada wrote:

• at first, you (or we) should introduce st_num_entries() (or something good name) to wrap to access num_entries/num_elements before your patch.

It seems unnecessary for me to rename num_entries.

Ok. I rethink the terminology to keep meaning the entries.

#8 - 03/04/2016 05:35 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

Great!

Thanks.

Notes:

• num_entries should remain num_entries. It is easier for you to change naming than fix all rubygems.

Thanks for pointing this out.

• do not change formatting of a code you do not change, it is awful to read and check that part of your patch.

I'll restore it. It is probably because I work on other projects which uses a different formatting.

• speed improvement is not from open addressing but from array storage for elements. You can use chaining with "next_index" with same result. But perhaps open addressing is simpler to implement.

Sorry, I doubt in this conclusion unless you prove it by benchmarks. Open addressing removes pointer making a smaller element which increases probability to be read from memory by reading the same cache line or/and probability to stay in a cache.

In case of collisions, I believe checking other entry again improves data locality in comparison with going by pointers through disperse elements (may be allocated originally for different tables).

On the other hand, you are right. I think the biggest improvement comes from putting elements into an array.

Open addressing is not simpler to implement too. The code might be smaller. For example, I suspect CPython has a wrong implementation of hash tables and can be cycled in extremely rare cases. For open addressing, someone should implement a full cycle linear congruential generator. I use $X_{next} = (5 \times X_{prev} + 1) \mod pow\_of\_2$ since it satisfies the requirements of the Hull-Dobell theorem. CPython function lookdict violates the theorem requirement $0 \leq X < 'the module' and consequently not a full cycle linear congruential generator. So implementing a correct open addressing is not easy. It is easy if you use prime numbers (that is what GCC
hash tables uses) but dividing big values (hashes) by prime numbers is even slower
(> 100 cycles) than access to main memory. By the way I tried using prime numbers
too (in average the implementation was faster than the current tables but
there were a few benchmarks where it was slower).

- if you stick with open-addressing, then it could be even faster, if you store hashsum in st_entry.

I tried something analogous and it did not work. Storing hash with entries increases
the table size as the array entries is bigger than the array elements.

- st_foreach is broken in case table were rebuilt.

Sorry, I did not catch what you meant. St_foreach works fine in the proposed implementation
even if the table rebuilds. Moreover st_foreach in the proposed implementation
can work forever adding and removing elements meanwhile the current implementation
will result out of memory in such case.

Any way, it is a great attempt I wished to do by myself, but didn't give time for. I hope, you'll fix all issues and change will be accepted.

I don't think the faster hash tables is that important for general Ruby speed.
Simply I need to start with some simpler project on MRI.

#9 - 03/04/2016 05:54 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

Small note: there is no need in perturbation and 'python like' secondary hash cause ruby uses strong hash function. Better stick with quadrating
probing.

Sorry, it is hard for me to understand why is it better.

Quadratic probing is most probably not faster on modern super-scalar OOO CPUs
than the secondary hash function I use. Quadratic probing will traverse all
entries for sure if # of entries is a prime number. As I wrote division of big
numbers (hash) by a prime number (for primary hash function) is > 100 cycles
according to the official Intel documents (but some researches say it is much more
than the official digits). So using prime numbers is out of question (I really
tried this approach too. It did not work). So for fast tables, the size should
be a power of 2.

As for not using perturbation. It means ignoring part of the hash (the biggest part
for most cases). I don't see it logical. Continuing this speculation, we could
generate 32-bit or even 16-bit hash instead of 64-bit as it is now.

#10 - 03/04/2016 06:13 PM - vmakarov (Vladimir Makarov)
Yusuke Endoh wrote:

Great, but it seems to need more work. It consumes nearly 2 times much memory.

# trunk
$ /usr/bin/time -f %Mkb ./miniruby -e '(0..1000000).map { { l => l } }'  
280108kb

# hash_tables_with_open_addressing branch
$ /usr/bin/time -f %Mkb ./miniruby -e '(0..1000000).map { { l => l } }'  
500264kb

Thanks for the numbers. Is it a real world scenario? I mean using huge
numbers of only small tables. I can imagine this when hash tables were used
for Ruby method tables. But it is not the case now.

May be I am wrong but I don't see how it helps with memory problems (mostly
GC) which dynamic languages experience.

But if you think that memory for such cases is important we can implement
the same approach as for the current hash tables when they are small. Simply
the code will be more complicated. I started this work once but I did not
like the code complication (the code might be a bit slower too as we need to check
in many places what variant of the table we use). Still I can implement it
if people think it is important. Please, just let me know overall people opinion about necessity of this.

#11 - 03/04/2016 11:11 PM - mame (Yusuke Endoh)

Vladimir Makarov wrote:

Thanks for the numbers. Is it a real world scenario?

I can imagine many cases: loading big data such as JSON and YAML, some kind of data structures such as Trie, data processing/aggregation, etc.

And I can confirm this problem not only with micro benchmark but also with an actual program. My local program (not published yet) consumes 76684kb without your patch, 106600kb with your patch. I don't think it uses Hash so heavily, though.

But if you think that memory for such cases is important we can implement the same approach as for the current hash tables when they are small.

It consumes more memory not only when they are small.

# trunk
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times{|n| h[n] = n }'
647512kb

# with your patch
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times{|n| h[n] = n }'
793844kb

To be fair: it runs 2 times faster with your patch. It is indeed great, but I don't think the memory consumption is negligible. It is a big trade-off.

# trunk
$ time ./miniruby -e 'h = {}; 10000000.times{|n| h[n] = n }'
real  0m4.885s
user  0m4.720s
sys   0m0.140s

# with your patch
$ time ./miniruby -e 'h = {}; 10000000.times{|n| h[n] = n }'
real  0m2.774s
user  0m2.668s
sys   0m0.092s

--
Yusuke Endoh mame@ruby-lang.org

#12 - 03/05/2016 01:25 AM - vmakarov (Vladimir Makarov)

Yusuke Endoh wrote:

Vladimir Makarov wrote:

Thanks for the numbers. Is it a real world scenario?

I can imagine many cases: loading big data such as JSON and YAML, some kind of data structures such as Trie, data processing/aggregation, etc.

And I can confirm this problem not only with micro benchmark but also with an actual program. My local program (not published yet) consumes 76684kb without your patch, 106600kb with your patch. I don't think it uses Hash so heavily, though.

Thanks for the cases.

OK. Then I should try to implement a compact representation of small tables even if it complicates the code.

But if you think that memory for such cases is important we can implement the same approach as for the current hash tables when they are small.

It consumes more memory not only when they are small.

# trunk
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times { |n| h[n] = n }'
 647512kb

# with your patch
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times { |n| h[n] = n }'
 793844kb

Thanks for the test. I'll investigate what is going on and try also other test parameters to see a pattern. It might be a difference in table rebuilding strategy.

#13 - 03/05/2016 03:37 AM - vmakarov (Vladimir Makarov)

Yusuke Endoh wrote:

It consumes more memory not only when they are small.

# trunk
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times { |n| h[n] = n }'
 647512kb

# with your patch
$ /usr/bin/time -f %Mkb ./miniruby -e 'h = {}; 10000000.times { |n| h[n] = n }'
 793844kb

I tried 4 different parameters for the test: 100000 1000000 10000000 100000000

The trunk ruby gives on my machine

15576kb
73308kb
650104kb
6520868kb

ruby with the proposed hash tables gives

15260kb
58268kb
795764kb
6300612kb

In 3 cases out of 4, the proposed hash tables are more compact than the trunk ones. So you were unfortunate with the test parameter.

It says for me the size of big tables is probably not an important problem.

But still sizes of small tables might be if it results in a slower code. Unfortunately it does. I confirm the slowdown problem exists for tables of size <= 3 for your test in the first email. So I will work on the small table problem.

Doing performance improvements is a hard job (I've been experiencing it well doing such job for GCC for last 20 years). There are always tests where the result can be worse. The average results (or even better a geometric mean) on credible benchmarks should be a major criterion. If there is requirement that everything should be better, there will be no progress on performance front after some short initial time of the project. I wrote this long paragraph here in order to make other people expectation more clear -- whatever I'd do with the tables, someone can find a test where the new tables will be worse on speed or/and memory consumption.

#14 - 03/05/2016 04:27 AM - mame (Yusuke Endoh)

In 3 cases out of 4, the proposed hash tables are more compact than the trunk ones.

Oops, I didn't realize. Sorry for the noise.

The average results (or even better a geometric mean) on credible benchmarks should be a major criterion.

Agreed. Can anybody test it with Rails?
Quadratic probing is most probably not faster on modern super-scalar OOO CPUs than the secondary hash function I use. Quadratic probing will traverse all entries for sure if # of entries is a prime number.

Looks like you didn’t do your homework :-(

https://en.m.wikipedia.org/wiki/Quadratic_probing

For \( m = 2^n \) a good choice for the constants are \( c1 = c2 = 1/2 \), as the values of \( h(k,i) \) for \( i \in [0,m-1] \) are all distinct. This leads to a probe sequence of \( h(k), h(k)+1, h(k)+3, h(k)+6, \ldots \) where the values increase by 1, 2, 3, ...

So usually it is implemented as:

```
pos = hash & mask;
d = 1;
while (not_found && not_empty) {
pos = (pos + d) & mask;
d++;
}
```

It probes all elements of table of size \( 2^n \) and has good cache locality for first few probes. So if you store 32bit hash sum there, it will be very fast to check.

Vladimir Makarov wrote:

Open addressing removes pointer making a smaller element which increases probability to be read from memory by reading the same cache line or/and probability to stay in a cache.

In case of collisions, I believe checking other entry again improves data locality in comparison with going by pointers through disperse elements (may be allocated originally for different tables).

I didn’t say to use pointers. I mean to use indices:

```c
typedef uint32_t st_index_t;
typedef st_index_t st_entry_t;

struct st_table_entry {
  st_index_t hash;
  st_index_t next;
  st_data_t key;
  st_data_t value;
};
```

Then you may use closed addressing with same effect as your open addressing. But instead of probing next element in entries array, you will probe it by next index. Cache locality will remain the same, cause anyway you’ll probe “random” entries. (unless you store hash in st_entry and use quadrating probing, which has better hash locality).

This way entries array could be smaller than elements entries. And small table may have no entries at all, just chain all elements in a hash smaller than \( n \) elements (\( n=6 \) for example).

st_foreach is broken in case table were rebuilt.

Sorry, I did not catch what you meant. St_foreach works fine in the proposed implementation even if the table rebuilds. Moreover st_foreach in the proposed implementation can work forever adding and removing elements meanwhile the current implementation will result out of memory in such case.

https://github.com/vnmakarov/ruby/blob/hash_tables_with_open_addressing/st.c#L927-L930
```c
if (rebuilds_num != tab->rebuilds_num) {
    elements = tab->elements;
    curr_element_ptr = &elements[i];
}
```

why do you believe that index i will be the same for this entry????

https://github.com/vnmakarov/ruby/blob/hash_tables_with_open_addressing/st.c#L442-L443

You skip deleted elements in a middle, so if there were deleted elements, i will be smaller after rebuild.

before rebuild:
  | 0 | 1 | 2 |
  | key=x | deleted | key=y |

after rebuild:
  | 0 | 1 |
  | key=x | key=y |

So if you position during iteration were on key=y (so i=2), then after rebuild it points to wrong place.

### #17 - 03/05/2016 10:40 AM - funny_falcon (Yura Sokolov)
I strongly believe st_index_t should be uint32_t. It will limit hash size to $2^{32}$ elements, but even with this "small" table it means that hash will consume at least 100GB of memory before limits reached.

### #18 - 03/05/2016 12:03 PM - funny_falcon (Yura Sokolov)
Next note: sizeof(struct st_table) were 48 bytes before patch on x86_64, and after it is 88 bytes (which will be rounded to 96 bytes by jemalloc).

Small hashes are very common despite your expectations, cause keyword arguments are passed as a hashes, and most methods has just a couple of arguments.

If you'll change st_index_t to uint32_t, then sizeof(struct st_table) will be 56 bytes.

```c
hash_mask == allocated_entries - 1, so no need to store.
```

If you'll use closed addressing, then no need in deleted_entries, and entries array could be not bigger than elements array.

With this two changes, sizeof(struct st_table) == 48.

### #19 - 03/05/2016 12:07 PM - funny_falcon (Yura Sokolov)
Still, rebuilding is very time consuming process. And there are applications when Hash is used as LRU cache.

Could you imagine that Hash with 1M elements starts to rebuild?

May be it is better to keep st_index_t prev, next in struct st_table_entry (or struct st_table_elements as you called it) ?

### #20 - 03/05/2016 05:50 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

> Quadratic probing is most probably not faster on modern super-scalar OOO CPUs than the secondary hash function I use. Quadratic probing will traverse all entries for sure if # of entries is a prime number.

Looks like you didn't do your homework :-(

No, I did not. Sorry. My memory failed me.

https://en.m.wikipedia.org/wiki/Quadratic_probing

For $m = 2^n$ a good choice for the constants are $c1 = c2 = 1/2$, as the values of $h(k,i)$ for $i$ in $[0, m-1]$ are all distinct. This leads to a probe sequence of $h(k)$, $h(k)+1$, $h(k)+3$, $h(k)+6$, ... where the values increase by 1, 2, 3, ...

So usually it is implemented as:

```c
pos = hash & mask;  
d = 1;  
while (not_found && not_empty) {  
    pos = (pos + d) & mask;  
d++;  
```
I believe your code above is incorrect for tables of sizes of power of 2. The function should look like $h(k,i) = (h(k) + c_1 i + c_2 i^2) \mod m$, where $c_1 = c_2 = 1/2$ is a good choice. You cannot simplify it. The same Wikipedia article contains

With the exception of the triangular number case for a power-of-two-sized hash table, there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not “prime”.

I don't see the quadratic function for sizes of power of 2 is simpler than what I use. It probes all elements of table of size $2^n$ and has good cache locality for first few probes. So if you store 32bit hash sum there, it will be very fast to check.

The only idea I like in your proposal is a better code locality argument. Also as I wrote before your proposal means just throwing away the biggest part of hash value even if it is a 32-bit hash. I don't think ignoring the big part of the hash is a good idea as it probably worsens collision avoiding. Better code locality also means more collision probability. However only benchmarking can tell this for sure. But I have reasonable doubts that it will be better.

Also about storing only part of the hash. Can it affect rubygems? It may be a part of API. But I don't know anything about it.

In any case trying your proposal is a very low-priority task for me (high priority task is a small table representation). May be somebody else will try it. It is not a wise approach to try it all and then stop. I prefer improvements step by step.

#21 - 03/05/2016 06:01 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

I strongly believe st_index_t should be uint32_t. It will limit hash size to $2^{32}$ elements, but even with this “small” table it means that hash will consume at least 100GB of memory before limits reached.

I routinely use a few machines for my development with 128GB memory. In a few years, there is a big probability of non-volatile memory with capacity of modern hard disks with access time comparable with the current memory and CPUs will have insns for access to it. At least some companies are working on these new technologies.

So opposite to you, I don't believe that we should make it 32-bit.

Besides I tried it. My goal was to make the array entry smaller for small or medium size tables in hope that it will improve the performance. Unfortunately, I did not see a visible performance improvement (just a noise) on the 21 hash table performance benchmarks on x86-64 when I made the index of 32-bit size.

I don't know how this thread from table speed discussion switched to memory usage discussion. Memory usage is important especially if it results in speed improvement. But I always considered it a secondary thing (although it was quit different when I started my programming on old super-computers with 32K 48-bit words).

#22 - 03/05/2016 06:04 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

Next note: sizeof(struct st_table) were 48 bytes before patch on x86_64, and after it is 88 bytes (which will be rounded to 96 bytes by jemalloc).

Small hashes are very common despite your expectations, cause keyword arguments are passed as a hashes, and most methods has just a couple of arguments.

OK. As I wrote I am going to implement a compact representation of small tables. This will have bigger effect on the size than the header optimization. The current table minimal hash size (a compact representation) as I remember is 48+6x3x8 = 192B. On my evaluation the minimal size of my tables will be 96 + 4x3x8 = 192B. So I guess I focus on small table compact representation instead of header optimizations as it really solves a slowdown problem for tables with less 4 elements which I wrote about recently.

If you'll change st_index_t to uint32_t, then sizeof(struct st_table) will be 56 bytes.

Please see my previous email about my opinion on this proposal.

    hash_mask == allocated_entries - 1, so no need to store.

I can and will do this. I missed this. That is what other implementations I mentioned do. Thanks for pointing this out.

    If you'll use closed addressing, then no need in deleted_entries, and entries array could be not bigger than elements array.
With this two changes, sizeof(struct st_table) == 48.

#23 - 03/05/2016 06:07 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

Still, rebuilding is very time consuming process. And there are applications when Hash is used as LRU cache.

Could you provide a test to investigate this.

Could you imagine that Hash with 1M elements starts to rebuild?

I can. The current tables do it all the time already and it means traversing all the elements as in the proposed tables case.

May be it is better to keep st_index_t prev, next in struct st_table_entry (or struct st_table_elements as you called it) ?

Sorry, I can not catch what do you mean. What prev, next should be used for. How can it avoid table rebuilding which always mean traversing all elements to find a new entry or bucket for the elements.

#24 - 03/05/2016 10:08 PM - funny_falcon (Yura Sokolov)
Vladimir, you acts as if i said a rubbish or i'm trying to cheat you. It makes me angry.

You wrote:

I believe your code above is incorrect for tables of sizes of power of 2.
The function should look like h(k,i) = (h(k) + c1 * i + c2 * i^2) mod m, where "c1 = c2 = 1/2 is a good choice". You can not simplify it.

And you cited Wikipedia

With the exception of the triangular number case for a power-of-two-sized hash table, there is no guarantee of finding an empty cell once the table gets more than half full

But couple of lines above you cited my cite from Wikipedia:

This leads to a probe sequence of h(k), h(k)+1, h(k)+3, h(k)+6, ...
where the values increase by 1, 2, 3, ...

Do you read carefully before answer?
It is implementation of triangular number sequence - single quadratic probing sequence which walks across all elements of 2^n table.

You even can remember arithmetic: https://en.wikipedia.org/wiki/Arithmetic_progression

\[(1/2)*i + (1/2)*i^2 = i*(i+1)/2 = 1 + 2 + 3 + ... + (i-1) + i\]

Or use Ruby to check you sentence:

2.1.5 :002 > p, d = 0, 1; 8.times.map{ a=p; p=(p+d)&7; d+=1; a} => [0, 1, 3, 6, 2, 7, 5, 4]
2.1.5 :008 > p = 0; 8.times.map{|i| a=p+ 0.5*i + 0.5*i*i; a.to_i&7} => [0, 1, 3, 6, 2, 7, 5, 4]

If you still don't believe me, read this: https://en.wikipedia.org/wiki/Triangular_number

Google Dense/Sparse hash uses this sequence with table of 2^n
https://github.com/sparsehash/sparsehash/blob/75ada1728b8b18ce80d4052b55b0d16cc6782f4b/dpcimplementation.html#L153
https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604daae/src/sparsehash/intermdensehashtable.h#L650-L651
https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604daae/src/sparsehash/intermdensehashtable.h#L119

khash uses quadratic probing now with table of 2^n

Please, check your self before saying other man is mistaken.
Or at least say with less level of confidence.

Also as I wrote before your proposal means just throwing away the biggest part of hash value even if it is a 32-bit hash. I don't think ignoring the big part of the hash is a good idea as it probably worsens collision avoiding.

Please about Birthday Paradox carefully. Yes, it will certainly increase probability of hash value collision, but only for very huge hash tables. And it doesn't affect length of a collision chain (cause 2^n tables uses only low bits). It just affects probability of excess call to equality check on value, but not too much:

http://math.stackexchange.com/a/35798

\[
N = (2^{32}).to_f
4294967296.0
\]
\[
n = 100_000_000.0
100000000.0
\]
\[
collisions = n*(1-(1-1/N)**(n-1))
2301410.50385877
\]
\[
collisions / n
0.0230141050385877
\]
\[
n = 300_000_000.0
100000000.0
\]
\[
collisions = n*(1-(1-1/N)**(n-1))
20239667.356876057
\]
\[
collisions / n
0.06746555785625352
\]

In other words, only 2% of full hash collisions on a Hash with 100_000_000_000 elements, and 7% for 300_000_000_000 elements. Can you measure, how much time will consume insertion of 100_000_000_000 elements to a Hash (current or your implementation), and how much memory it will consume? Int=>Int? String=>String?

At my work, we use a huge in-memory hash tables (hundreds of millions elements) (custom in-memory db, not Ruby), and it uses 32bit hashsum. No any problem. At this
gems ought to be recompiled, but no code change.

I routinely use a few machines for my development with 128GB memory.

But you wouldn't use Ruby process which consumes 100GB of memory using Ruby Hash. Otherwise you get a big trouble (with GC for example). If you need to store such amount of data within Ruby Process, you'd better make your own datastructure.

I've maid one for my needs:
https://rubygems.org/gems/inmemory_kv
https://github.com/funny-falcon/inmemory_kv
It also can store only 2^31 elements, but hardly beleive you will ever store more inside of Ruby process.

Could you imagine that Hash with 1M elements starts to rebuild?
I can. The current tables do it all the time already and it means traversing all the elements as in the proposed tables case.

Current st_table rebuilds only if its size grow. Your table will rebuild even if size is not changed much, but elements are inserted and deleted repeatedly (1 add, 1 delete, 1 add, 1 delete)

May be it is better to keep st_index_t prev, next in struct st_table_entry (or struct st_table_elements as you called it)?
Sorry, I can not catch what you mean. What prev, next should be used for.
How can it avoid table rebuilding which always mean traversing all elements to find a new entry or bucket for the elements.

Yeah, it is inevitable to maintain free list for finding free element.
But prev,next indices will allow to insert new elements in random places (deleted before), cause iteration will go by this pseudo-pointers.

Perhaps, it is better to make a separate LRU hash structure instead in a standard library, and keep Hash implementation as you suggest.
I really like this case, but it means Ruby will have two hash tables - for Hash and for LRU.

#25 - 03/05/2016 10:36 PM - funny_falcon (Yura Sokolov)
Vladimir, I'm sorry, I am too emotional.
Yura Sokolov wrote:

Vladimir, I'm sorry, I am too emotional.

That is OK.

Encouraging performance results!

I didn't test fully, but "make test-all" gets stuck for me

```
make test-all TESTS=test/ruby/test_hash.rb
```

I'm on Debian wheezy, x86-64 gcc-4.7.real (Debian 4.7.2-5) 4.7.2

There may be other tests which fail, but I didn't investigate further.

Other notes (mostly reiterating other comments):

In my experience, more users complain about memory usage than performance since Ruby 1.9/2.x days. Maybe it's because traditionally, the Ruby ecosystem did not have great non-blocking I/O or thread support; users find it easier to fork processes, instead.

Small hashes are common in Ruby and important to parameter passing, ivar indices in small classes, short-lived statistics, and many other cases. Because hashes are so easy-to-create, Rubyists tend to create many of them.

```
st.h is unfortunately part of our public C API; so num_entries shouldn't change. I propose we hide the new struct fields somehow in similar fashion to private_list_head or at least give them scary names which discourage public use.
```

Anyways, I'm excited about these changes and hope we can get the most of the benefits without the downsides.

Yura Sokolov wrote:

Vladimir, you acts as if i said a rubbish or i'm trying to cheat you. It makes me angry.

You wrote:

I believe your code above is incorrect for tables of sizes of power of 2.
The function should look like $h(k,i) = (h(k) + c1 \cdot i + c2 \cdot i^2) \mod m$,
where "c1 = c2 = 1/2 is a good choice". You can not simplify it.

And you cited Wikipedia

With the exception of the triangular number case for a power-of-two-sized hash table, there is no guarantee of finding an empty cell once the table gets more than half full

But couple of lines above you cited my cite from Wikipedia:

This leads to a probe sequence of $h(k), h(k)+1, h(k)+3, h(k)+6, ...$
where the values increase by 1, 2, 3, ...

Do you read carefully before answer?
It is implementation of triangular number sequence - single quadratic probing sequence which walks across all elements of $2^n$ table.

You even can remember arithmetic: [https://en.wikipedia.org/wiki/Arithmetic_progression](https://en.wikipedia.org/wiki/Arithmetic_progression)

\[
\frac{1}{2}i + \frac{1}{2} + \frac{1}{2}i^2i = i*(i+1)/2 = 1 + 2 + 3 + ... + (i-1) + i
\]
Or use Ruby to check you sentence:

```ruby
2.1.5 :002 > p, d = 0, 1; 8.times.map{ a=p; p=(p+d)&7; d+=1; a}
=> [0, 1, 3, 6, 2, 7, 5, 4]
```

If you still don't believe me, read this: [https://en.wikipedia.org/wiki/Triangular_number](https://en.wikipedia.org/wiki/Triangular_number)

Google Dense/Sparse hash uses this sequence with table of 2^n
[https://github.com/sparsehash/sparsehash/blob/75ada1728b8b18ce80d4052b55b0d16cc6782f4b/doc/implementation.html#L153](https://github.com/sparsehash/sparsehash/blob/75ada1728b8b18ce80d4052b55b0d16cc6782f4b/doc/implementation.html#L153)
[https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604dae/src/sparsehash/internal/densehashtable.h#L65-L651](https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604dae/src/sparsehash/internal/densehashtable.h#L65-L651)
[https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604dae/src/sparsehash/internal/densehashtable.h#L119](https://github.com/sparsehash/sparsehash/blob/a61a6ba7adbc4e3a7545843a72c530bf35604dae/src/sparsehash/internal/densehashtable.h#L119)

khash uses quadratic probing now with table of 2^n

Please, check your self before saying other man is mistaken. Or at least say with less level of confidence.

I am really sorry. People do mistakes. I should have not written this in a hurry while running some errands.

Still I can not see how using the function \((p + d) \& h\) instead of \((i \ll 2 + i + p + 1) \& m\) visibly speedups hash tables (which is memory bound code) on modern OOO super-scalar CPUs. That is besides advantages (even tiny as you are writing below) of decreasing collisions by using all hash.

Actually I did an experiment. I tried these two functions on Intel Haswell with memory access to the same address (so the value will be in the 1st cache) after each function calculation. I used -O3 for GCC and run the functions 10^9 times. The result is a bit strange. Code with function \((i \ll 2 + i + p + 1) \& m\) is about 7% faster than one with simpler function \((p + d) \& h\) (14.5s vs. 15.7s). Sometimes it is hard to predict outcome as modern x86-64 processors are black boxes and actually interpreters inside. But even if it was the opposite, absence of value in the cache and the fact, that the function is a small part of code for the access by a key, probably will make that difference insignificant.

Also as I wrote before your proposal means just throwing away the biggest part of hash value even if it is a 32-bit hash. I don't think ignoring the big part of the hash is a good idea as it probably worsens collision avoiding.


Yes, it will certainly increase probability of hash value collision, but only for very HUGE hash tables. And it doesn't affect length of a collision chain (cause 2^n tables uses only low bits).

It just affects probability of excess call to equality check on value, but not too much:

[http://math.stackexchange.com/a/35798](http://math.stackexchange.com/a/35798)

In other words, only 2% of full hash collisions on a Hash with 100,000,000 elements, and 7% for 300,000,000 elements. Can you measure, how much time will consume insertion of 100,000,000 elements to a Hash (current or your implementation), and how much memory it will consume? Int=>Int ? String=>String?

On a machine currently available to me,

```
./ruby -e 'h = {}; 100_000_000.times {|n| h[n] = n}''
```

takes 7m for my implementation.

The machine has not enough memory for 300,000,000 elements. So I did not try.

In GCC community where I am from, we are happy if we all together improved SPEC2000/SPEC2006 by 1%-2% during a year. So if I can use all...
hash without visible slowdown even if it decreases number of collisions only by 1% on big tables, I'll take that chance.

At my work, we use a huge in-memory hash tables (hundreds of millions elements) (custom in-memory db, not Ruby), and it uses 32bit hashsum. No any problem. At this

Also about storing only part of the hash. Can it affect rubygems? It may be a part of API. But I don't know anything about it
gems ought to be recompiled, but no code change.

Thanks for the answer.

I routinely use a few machines for my development with 128GB memory.

But you wouldn't use Ruby process which consumes 100GB of memory using Ruby Hash. Otherwise you get a big trouble (with GC for example).
If you need to store such amount of data within Ruby Process, you'd better make your own datastructure.
I've maid one for my needs:
https://rubygems.org/gems/inmemory_kv
https://github.com/funny-falcon/inmemory_kv
It also can store only 2^31 elements, but hardly beleive you will ever store more inside of Ruby process.

IMHO, I think it is better to fix the problems which can occur for using tables with > 2^32 elements instead of introducing the hard constraint. There are currently machines which have enough memory to hold tables with > 2^32 elements. You are right it is probably not wise to use MRI to work with such big tables regularly because MRI is currently slow for this. But it can be used occasionally for prototyping. Who knows may be MRI will become much faster.

But my major argument is that using 32-bit index does not speed up work with hash tables. As I wrote I tried it and using 32-bit index did not improve the performance. So why we should create such hard constraint then.

Could you imagine that Hash with 1M elements starts to rebuild?
I can. The current tables do it all the time already and it means traversing all the elements as in the proposed tables case.

Current st_table rebuilds only if its size grow. Your table will rebuild even if size is not changed much, but elements are inserted and deleted repeatedly (1 add, 1 delete, 1 add, 1 delete)

May be it is better to keep st_index_t prev, next in struct st_table_entry (or struct st_table_elements as you called it) ?
Sorry, I can not catch what you mean. What prev, next should be used for.
How can it avoid table rebuilding which always mean traversing all elements to find a new entry or bucket for the elements.

Yeah, it is inevitable to maintain free list for finding free element.
But prev,next indices will allow to insert new elements in random places (deleted before),
cause iteration will go by this pseudo-pointers.

There is no rebuilding if you use hash tables as a stack. The same can be also achieved for a queue with some minor changes. As I wrote there will be always a test where a new implementation will behave worse. It is a decision in choosing what is better: say 50% improvement on some access patterns or n% improvement on other access patterns. I don't know how big is n. I tried analogous approach what you proposed. According to my notes (that time MRI has only 17 tests), the results were the following

<table>
<thead>
<tr>
<th>name</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_aref_dsym</td>
<td>0.811</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.360</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>0.744</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.123</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>0.811</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>0.836</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>0.896</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>0.847</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.149</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.730</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.812</td>
</tr>
<tr>
<td>name</td>
<td>time</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.765</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.797</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.807</td>
</tr>
<tr>
<td>hash_keys</td>
<td>1.456</td>
</tr>
<tr>
<td>hash_shift</td>
<td>0.038</td>
</tr>
<tr>
<td>hash_values</td>
<td>1.450</td>
</tr>
</tbody>
</table>

But unfortunately they are not representative as I used prime numbers as size of the tables and just mod for mapping hash value to entry index.

Perhaps, it is better to make a separate LRU hash structure instead in a standard library, and keep Hash implementation as you suggest. I really like this case, but it means Ruby will have two hash tables - for Hash and for LRU.

I don't know.

In any case, it is not up to me to decide the size of the index and some other things discussed here. That is why probably I should have not participated in this discussion.

We spent a lot time arguing. But what should we do is trying. Only real experiments can prove or disapprove our speculations. May be I'll try your ideas but only after adding my code to MRI trunk. I still have to solve the small table problem people wrote me about before doing this.

#29 - 03/06/2016 04:26 AM - vmakarov (Vladimir Makarov)

Eric Wong wrote:

Encouraging performance results!

Thanks.

I didn't test fully, but "make test-all" gets stuck for me

test_calloc of TestHash:

make test-all TESTS=test/ruby/test_hash.rb

I'm on Debian wheezy, x86-64 gcc-4.7.real (Debian 4.7.2-5) 4.7.2
There may be other tests which fail, but I didn't investigate further.

I am a novice to MRI. When I used test-all recently I have some errors on the trunk with and without my code. So I used only test. I guess I should try what you did.

Other notes (mostly reiterating other comments):

In my experience, more users complain about memory usage than performance since Ruby 1.9/2.x days. Maybe it's because traditionally, the Ruby ecosystem did not have great non-blocking I/O or thread support; users find it easier to fork processes, instead.

Interesting. I got a different impression that people complaining more about that MRI is slow. Moving to VM was a really great improvement (2-3 times as I remember) and made Ruby actually faster than CPython. But it is still much slower PyPy, JS, LuaJIT. That is why probably people are complaining. Getting this impression I decided to try to help improving MRI performance. MRI is the language definition so I think working on alternative Ruby implementations would be not wise. I have some ideas and hope my management permits me to spend part of my time to work on their implementation.

Small hashes are common in Ruby and important to parameter passing, ivar indices in small classes, short-lived statistics, and many other cases. Because hashes are so easy-to-create, Rubyists tend to create many of them.

Thanks. I completely realize now that compact hash tables are important.

I propose we hide the new struct fields.
somehow in similar fashion to private_list_head or at least give them scary names which discourage public use.

Yes, I'll fix it and use the old names or try to figure out a better terminology which does not change names in st.h.

Anyways, I'm excited about these changes and hope we can get the most of the benefits without the downsides.

Thanks again.

#30 - 03/06/2016 08:06 AM - funny_falcon (Yura Sokolov)

Code with function \((i << 2 + i + p + 1) \& m\) is about 7% faster than one with simpler function \((p + d) \& h\) (14.5s vs. 15.7s)

Don't you forget that \(d\) should be incremented every step? Otherwise it is linear probing instead of quadratic probing.

So if I can use all hash without visible slowdown even if it decreases number of collisions only by 1% on big tables, I'll take that chance.

But my major argument is that using 32-bit index does not speed up work with hash tables. As I wrote I tried it and using 32-bit index did not improve the performance. So why we should create such hard constraint then.

If you stick with open addressing, then storing 32bit hash in 'entries' array and using 32bit index may speedup hash.

If you try to closed addressing, then 32bit hash + 32bit next pointer will allow to not increase element size.

Prototyping with \(2^{31}\) elements in memory?
I work in small social network (monthly auditory 30M, daily 5M), that exists for 9 years, and some in-memory tables sharded to 30 servers just get close to \(2^{30}\) totally (i.e. sum among all servers).

Please test time for inserting 200_000_000, 300_000_000 elements, does the time grows lineary?
And you didn't measure String keys or/and values.
You said, you computer has memory for 300_000_000 elements, how much it is? How much memory will 1_000_000_000 Int=>Int elements will consume? How much 1_000_000_000 memory String=>String will consume?

Concerning prev,next

I tried analogous approach what you proposed.

hash_shift 0.038

hash_shift result shows that your implementation had flaws, so performance numbers are not representable. I do not expect performance change lesser than 0.95.

What if it is not LRU? What if it is general purpose 100_000_000 table?
You said a lot of about 1% improvement on such table.
What will you say about excessive 2 second pause for rebuilt such table?
How much pause will be for 1_000_000_000 table?

#31 - 03/06/2016 08:24 AM - funny_falcon (Yura Sokolov)

7 minutes for filling 100_000_000 table (Int=>Int).
Pretending time grows lineary, 140 minutes for filling 2_000_000_000 table.
And useful work is at least twice time more: +280=420 minutes.

7 hours to test prototype? And it is just Int=>Int! With strings it will be several times more.

At this level people switches to Go, Closure or something else.

#32 - 03/06/2016 02:28 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

7 minutes for filling 100_000_000 table (Int=>Int).
Pretending time grows lineary, 140 minutes for filling 2_000_000_000 table.
And useful work is at least twice time more: +280=420 minutes.

7 hours to test prototype? And it is just Int=>Int! With strings it will be several times more.

At this level people switches to Go, Closure or something else.
Yura, after reading your last two emails my first impulse was to answer them. But doing this I've realized that I need to repeat my arguments third or second time, do some experiments trying to figure out why the test for big parameters is slow on Ruby, thinking how it can be fixed in MRI or on language level. This discussion became unproductive for me and wasting my time. The direction of the discussion is just again confirming to me that I should have not participated in this discussion.

I did not change my opinion, you did not change yours. As I wrote it is not up to me to decide what size of index and hash (32-bit or 64-bit on 64-bit machines) we should use. I don't know how such decisions are made in Ruby/MRI community. If the community decides to use 32-bit indexes and hashes, I am ready to change my code. You just need to convince now the decision makers to do such change. As for me I am stopping to discuss these issues. I am sorry for doing this.

#33 - 03/06/2016 03:01 PM - shyouhei (Shyouhei Urabe)

Very impressive work. I'm excited to see your patch boosting ruby.

Almost everything I thought was already said by others; what remains curious to me is that it seems a long-lasting hash objects tends to leak memory with your patch.

    zsh % for i in 1 10 100 1000 10000 100000 1000000 10000000
    zsh > do /usr/bin/time -f%Mkb ./miniruby -e "h=Hash.new; $i.times{|i|h[i] = i; h.delete(i-1)}"
    zsh > done
    3476kb
    3484kb
    3492kb
    3540kb
    3976kb
    7528kb
    36920kb
    497868kb
    zsh %

When you want to use a hash as an in-memory key-value store, it is quite natural for it to experience lots of random additions / deletions. I think this situation happens in real-world programs. Is this intentional or just a work-in-progress?

I'm also unsure about worst-case time complexity of your strategy. At a glance it seems not impossible for an attacker to intentionally collide entries (hope I'm wrong). On such situations, old hash we had did rebalancing called rehash to avoid long chains. Is there anything similar, or you don't have to, or something else?

Anyways, this is a great contribution. I wish I could do this. I hope you revise the patch to reflect other discussions and get it merged.

#34 - 03/06/2016 06:37 PM - vmakarov (Vladimir Makarov)

Shyouhei Urabe wrote:

    Very impressive work. I'm excited to see your patch boosting ruby.

Thanks.

Almost everything I thought was already said by others; what remains curious to me is that it seems a long-lasting hash objects tends to leak memory with your patch.

    zsh % for i in 1 10 100 1000 10000 100000 1000000 10000000
    zsh > do /usr/bin/time -f%Mkb ./miniruby -e "$i.times{|i|h[i] = i; h.delete(i-1)}"
    zsh > done
    3476kb
    3484kb
    3492kb
    3540kb
    3976kb
    7528kb
    36920kb
    497868kb
    zsh %

I don't think it is a leak. What you measure is the maximal residential size. I think the table is rebuilt many times and memory for previous version of tables is freed but it is not freed to OS by MRI (or glibc. I don't know yet what allocation library is used by MRI). Still this is very bad. I should definitely to investigate and fix it. I believe I know how to fix it. I should reuse the array elements when it is possible. Thanks for pointing this out.

Would you mind if I include your test in the final version of the patch as a benchmark?

    When you want to use a hash as an in-memory key-value store, it is quite natural for it to experience lots of random additions / deletions. I think this situation happens in real-world programs. Is this intentional or just a work-in-progress?
The proposed hash tables will work with random additions/deletions. I only did not know what the exact performance will be in comparison with the current tables. As I became aware of your case now (Yura Sokolov also wrote about it) it will be a work-in-progress for me.

I am not sure your case is real world case scenario (removing the last element is) but it is definitely the worst case for my implementation.

I'm also unsure about worst-case time complexity of your strategy. At a glance it seems not impossible for an attacker to intentionally collide entries (hope I'm wrong). On such situations, old hash we had did rebalancing called rehash to avoid long chains. Is there anything similar, or you don't have to, or something else?

The worst case is probably the same as for the current tables. It is theoretically possible to create test data which results in usage the same entry for the current and proposed tables. But in practice it is impossible for medium size table even if murmur hash is not a cryptography level hash function as, for example, sha2.

I've specifically chosen a very small hash table load (0.5) to make chance of collisions very small and rebuilding less frequent (such parameter still results in about the same non-small hash table sizes). I think even maximal load 3/4 would work well to avoid collision impact. People can make experiments with such parameters and change them later if something works better of course if the proposed tables will be in the trunk.

But still if there are a lot of collisions the same strategy can be used -- table rebuilding. I'll think about this.

Anyways, this is a great contribution. I wish I could do this. I hope you revise the patch to reflect other discussions and get it merged.

Yes, I'll work on the revised patch. The biggest change I see it right now will be a compact representation of small tables. Unfortunately, I can not promise the revised patch will be ready soon as I am busy with fixing bugs for GCC6. But this work on GCC6 will be finished in a month.

Thanks for your useful message.

#35 - 03/06/2016 07:01 PM - ko1 (Koichi Sasada)
On 2016/03/07 3:37, vmakarov@redhat.com wrote:

I don't think it is a leak. What you measure is the maximal residential size. I think the table is rebuilt many times and memory for previous version of tables is freed but it is not freed to OS by MRI (or glibc. I don't know yet what allocation library is used by MRI). Still this is very bad. I should definitely to investigate and fix it. I believe I know how to fix it. I should reuse the array elements when it is possible. Thanks for pointing this out.

trunk (without your patch):

```
$ for i in 1 10 100 1000 10000 100000 1000000 10000000
do /usr/bin/time -f%Mkb ./miniruby -e "h=Hash.new; $i.times{|i|h[i] = i; h.delete[i-1]}"
done
3504kb
3504kb
3512kb
3508kb
3504kb
3516kb
3516kb
3516kb
```

This is what I asked in my comment.

- (rare case) so many deletion can keep spaces (does it collected? i need to read code more)

We can generalize the last issue as "compaction". This is what I didn't touch this issue yet (maybe not a big problem).

And you didn't response about that :)

#36 - 03/06/2016 07:31 PM - ko1 (Koichi Sasada)
On 2016/03/05 1:31, vmakarov@redhat.com wrote:
So the packed element approach could be implemented too for the proposed implementation.

I agree.

I don't see it is necessary unless the hash tables will be again used for method tables where most of them are small.

As some people said, there are many small Hash objects, like that:

```ruby
def foo(**opts)
do_something opts[:some_option] || default_value
end

foo(another_option: customized_value)
```

BTW, from Ruby 2.2, most of passing keyword parameters does not create Hash object. In above case, a hash object is created explicitly (using ** a keyword hash parameter).

Hash tables will be faster than the used binary search. But it is not a critical code (at least for benchmarks in MRI) as we search method table once for a method and all other calls of the method skips this search. I am sure you know it much better.

Maybe we continue to use id_table for method table, or something. It is specialized for ID key table.

BTW (again), I (intuitively) think linear search is faster than using Hash table on small elements. We don't need to touch entries table. (But no evidence I have.)

For example, assume 8 elements. One element consume 24B, so that we need to load 8 * 24B = 192B on worst case (no entry) with linear search, 3 L1 cache misses on 64B L1 cache CPU.

However, if we collect hash values (making a hash value array), we only need to load 8 * 8B = 64B.

... sorry, it is not simple :p

Speaking of measurements. Could you recommend credible benchmarks for the measurements. I am in benchmarking business for a long time and I know benchmarking may be an evil. It is possible to create benchmarks which prove opposite things. In compiler field, we use SPEC2000/SPEC2006 which is a consensus of most parties involved in the compiler business. Do Ruby have something analogous?

as other people, i agree. and Ruby does not have enough benchmark :( I think discourse benchmark can help.

In the proposed implementation, the table size can be decreased. So in some way it is collected.

Reading the responses to all of which I am going to answer, I see people are worrying about memory usage. Smaller memory usage is important for better code locality too (although a better code locality does not mean a faster code automatically -- the access patter is important too). But I consider the speed is the first priority these days (especially when memory is cheap and it will be much cheaper with new coming memory technology).

In many cases speed is achieved by methods which requires more memory. For example, Intel compiler generates much bigger code than GCC to achieve better performance (this is most important competitive advantage for their compiler).

Case by case.
For example, Heroku smallest dyno only provides 512MB.
I think goods overcomes bads.

Thanks, I really appreciate your opinion. I'll work on the found issues. Although I am a bit busy right now with work on GCC6 release. I'll have more time to work on this in April.

So great.
I hope GCC6 released in success.


Yes, the term is confusing but it was used since 1957 according to Knuth.

I need to complain old heroes.

--
// SASADA Koichi at atdot dot net

#37 - 03/07/2016 01:05 AM - funny_falcon (Yura Sokolov)


So, Java HashMap and Hashtable have maximum size $2^{31}$. Does Ruby need more?

#38 - 03/07/2016 02:35 AM - vmakarov (Vladimir Makarov)

Koichi Sasada wrote:

On 2016/03/07 3:37, vmakarov@redhat.com wrote:

I don't think it is a leak. What you measure is the maximal residential size. I think the table is rebuilt many times and memory for previous version of tables is freed but it is not freed to OS by MRI (or glibc. I don't know yet what allocation library is used by MRI). Still this is very bad. I should definitely to investigate and fix it. I believe I know how to fix it. I should reuse the array elements when it is possible. Thanks for pointing this out.

trunk (without your patch):

$ for i in 1 10 100 1000 10000 100000 1000000 10000000
do /usr/bin/time -f%Mkb ./miniruby -e "h=Hash.new; $i.times{|i|h[i] = i; h.delete(i-1)}"done
3504kb
3504kb
3512kb
3516kb
3504kb
3508kb
3504kb
3516kb
3516kb
3516kb
3516kb

This is what I asked in my comment.

Thank you. I tried this too and see it is a real problem I should solve.

- (rare case) so many deletion can keep spaces (does it collected? i need to read code more)

... 

We can generalize the last issue as “compaction”.
This is what I didn't touch this issue yet (maybe not a big problem).

And you didn't response about that ;)

No, I did not. Sorry. I answered about another way how the proposed tables can reclaim memory. Now understanding what you meant I can say that there is no compaction. But you are already know it :)

05/08/2021
My thought behind skipping compaction was that in any case we should traverse all elements and it will take practically the same time as for table rebuilding while this approach simplifies code considerably. Getting responses from people providing small tests I see that I was wrong. I missed the effect of memory allocation layer. So we should definitely have the compaction.

Another thought I got recently that the compaction traversing even all elements have more chances to work with cached data than in the case of rebuilding.

---

#39 - 03/07/2016 02:41 AM - vmakarov (Vladimir Makarov)

Koichi Sasada wrote:

On 2016/03/05 1:31, vmakarov@redhat.com wrote:

So the packed element approach could be implemented too for the proposed implementation.

I agree.

I don't see it is necessary unless the hash tables will be again used for method tables where most of them are small.

As some people said, there are many small Hash objects, like that:

```ruby
def foo(**opts)
do_something opts[:some_option] || default_value
end

foo(another_option: customized_value)
```

BTW, from Ruby 2.2, most of passing keyword parameters does not create Hash object. In above case, a hash object is created explicitly (using ** a keyword hash parameter).

To be honest, I did not know that some parameters are passed as hash objects. I am not a Ruby programmer but I am learning.

Hash tables will be faster than the used binary search. But it is not a critical code (at least for benchmarks in MRI) as we search method table once for a method and all other calls of the method skips this search. I am sure you know it much better.

Maybe we continue to use id_table for method table, or something. It is specialized for ID key table.

BTW (again), I (intuitively) think linear search is faster than using Hash table on small elements. We don't need to touch entries table. (But no evidence I have.)

For example, assume 8 elements.

One element consume 24B, so that we need to load 8 * 24B = 192B on worst case (no entry) with linear search. 3 L1 cache misses on 64B L1 cache CPU.

However, if we collect hash values (making a hash value array), we only need to load 8 * 8B = 64B.

... sorry, it is not simple 😞

I am agree it is not simple. Especially if we take the effect of execution of other parts of the program on caches (how some code before and after and parallerly working with given code accesses memory). In general the environment effect can be important. For example, I read a lot research papers about compiler optimizations. They always claim big or modest improvements. But when you are trying the same algorithm in GCC environment (not in a toy compiler) the effect is frequently smaller, and even in rare cases the effect is opposite (a worse performance). Therefore, something affirmative can be said only after the final implementation.

If I correctly understood MRI VM code, you need to search id_table element usually once and after that the found value is stored in corresponding VM call insn. The search is done again if an object method or a class method is changed. Although this pattern ("monkey patching") exists, I don't consider it frequent. So I conclude the search in the id table is not critical (may be I am wrong).

For non-critical code, on my opinion the best strategy is to minimize cache changes for other parts of a program. With this point of view, linear or binary search is probably a good approach as the used data structure has a minimal footprint.
Speaking of measurements. Could you recommend credible benchmarks for the measurements. I am in benchmarking business for a long time and I know benchmarking may be an evil. It is possible to create benchmarks which prove opposite things. In compiler field, we use SPEC2000/SPEC2006 which is a consensus of most parties involved in the compiler business. Do Ruby have something analogous?

as other people, i agree. and Ruby does not have enough benchmark :( I think discourse benchmark can help.

As I know there are a lot of applications written on Ruby. May be it is possible to adapt a few of non IO- or network bound programs for benchmarking. It would be really useful. The current MRI benchmarks are micro-benchmarks they don't permit to see a bigger picture. Some RedHat people recommended for me to use fluentd for benchmarking. But I am not sure about this.

In the proposed implementation, the table size can be decreased. So in some way it is collected.

Reading the responses to all of which I am going to answer, I see people are worrying about memory usage. Smaller memory usage is important for better code locality too (although a better code locality does not mean a faster code automatically -- the access pattern is important too). But I consider the speed is the first priority these days (especially when memory is cheap and it will be much cheaper with new coming memory technology).

In many cases speed is achieved by methods which requires more memory. For example, Intel compiler generates much bigger code than GCC to achieve better performance (this is most important competitive advantage for their compiler).

Case by case. For example, Heroku smallest dyno only provides 512MB.

I think goods overcomes bads.

Thanks, I really appreciate your opinion. I'll work on the found issues. Although I am a bit busy right now with work on GCC6 release. I'll have more time to work on this in April.

So great. I hope GCC6 released in success.

- I always confuse about "open addressing" == "closed hashing" https://en.wikipedia.org/wiki/Open_addressing

Yes, the term is confusing but it was used since 1957 according to Knuth.

I need to complain old heroes.

Thank you for providing some new and useful info to me.

#40 - 03/07/2016 05:17 AM - shyouhei (Shyouhei Urabe)

Vladimir Makarov wrote:

I don't think it is a leak. What you measure is the maximal residential size. I think the table is rebuilt many times and memory for previous version of tables is freed but it is not freed to OS by MRI (or glibc. I don't know yet what allocation library is used by MRI). Still this is very bad. I should definitely to investigate and fix it. I believe I know how to fix it. I should reuse the array elements when it is possible. Thanks for pointing this out.

OK. Looking forward to the fix.

Would you mind if I include your test in the final version of the patch as a benchmark?

No problem. Do so please.
When you want to use a hash as an in-memory key-value store, it is quite natural for it to experience lots of random additions / deletions. I think this situation happens in real-world programs. Is this intentional or just a work-in-progress?

The proposed hash tables will work with random additions/deletions. I only did not know what the exact performance will be in comparison with the current tables. As I became aware of your case now (Yura Sokolov also wrote about it) it will be a work-in-progress for me.

I am not sure your case is real world case scenario (removing the last element is) but it is definitely the worst case for my implementation.

(WIP is definitely OK with me.) Let me think of a more realistic use case.

Often, a key-value store comes with expiration of keys. This is typically done by storing expiration time in each hash elements, then periodically call st_foreach() with a pointer argument to an expiration-checking function. To simulate this, I wrote following snippet:

```ruby
h = Hash.new
m = Mutex.new

# Thread to purge expired entries
Thread.start{
  loop{
    t = Time.now
    m.synchronize{
      h.reject!{|k, v| v < t }
    }
  }
}

# insertions
1000.times{|i|}
  t = Time.now + 0.1
  m.synchronize {
    1000.times{|j|
      h[1000*i+j] = t
    }
  }
  STDERR.puts "%d: %d\r", i, h.size
} puts
```

(The printf was for me to check progress.)

It ran with your patch and resulted in 37,668kb max resident memory. Without the patch, it took 8,988kb.

Above might still be a bit illustrative, but I believe this situation happens in wild.

The worst case is probably the same as for the current tables. It is theoretically possible to create test data which results in usage the same entry for the current and proposed tables. But in practice it is impossible for medium size table even if murmur hash is not a cryptography level hash function as, for example, sha2.

I've specifically chosen a very small hash table load (0.5) to make chance of collisions very small and rebuilding less frequent (such parameter still results in about the same non-small hash table sizes). I think even maximal load 3/4 would work well to avoid collision impact. People can make experiments with such parameters and change them later if something works better of course if the proposed tables will be in the trunk.

But still if there are a lot of collisions the same strategy can be used -- table rebuilding. I'll think about this.

Yes. Please do. "People are not evil(smart) enough to do this" -kind of assumptions tends to fail.
- Closed addressing + double linked list for insertion order
- 32bit indices instead of pointers and 32bit hashes
- Some tricks to reduce size of st_table, and fit array into sizes comfortable for allocators

(Patch attached, and also here https://github.com/ruby/ruby/compare/trunk...funny-falcon:st_table_with_array_storage.patch)

First number - trunk, second - Vladimirs, third - mine

```
hash_aref_dsym  0.558  0.556  0.563
hash_aref_dsym_long  12.049  7.784  6.841
hash_aref_fix  0.592  0.621  0.602
hash_aref_flo  0.15  0.13  0.135
hash_aref_miss  0.804  0.772  0.806
hash_aref_str  0.738  0.725  0.748
hash_aref_sym  0.551  0.552  0.556
hash_aref_sym_long  0.792  0.777  0.785
hash_flatten  0.546  0.498  0.546
hash_ident_flo  0.113  0.11  0.114
hash_ident_num  0.533  0.564  0.548
hash_ident_obj  0.523  0.529  0.53
hash_ident_str  0.527  0.529  0.543
hash_ident_sym  0.558  0.552  0.561
hash_keys  0.533  0.311  0.423
hash_shift  0.08  0.077  0.075
hash_shift_u16  0.176  0.202  0.148
hash_shift_u24  0.175  0.212  0.143
hash_shift_u32  0.172  0.203  0.143
hash_to_proc  0.066  0.059  0.064
hash_values  0.46  0.309  0.441
vm2_bighash  6.196  3.555  2.524
```

Memory usage is lesser than Vladimirs', and comparable to trunk (sometimes lesser than trunk, sometimes bigger). Usage may be bigger cause array is preallocated with room of empty elements. My patch has ability to specify more intermediate steps of increasing capacities, but allocator behavior should be considered.

So, as I said, main advantage is from storing st_table_entry as an array, so less calls to malloc performed, less TLB misses and less memory used. Open addressing gives almost nothing to performance in this case, cause it is not a case where open addressing plays well.

Open addressing plays well when you whole key-value structure is small and stored inside of hash-array. But in case of Ruby's Hash we store st_table_entry outside of open-addressing array, so jump is performed, and main benefit (cache locality) is lost.

Vladimir's proposal for storing insertion order by position in array can still benefit in memory usage, if carefully designed.

By the way, PHP's array do exactly this (but uses closed addressing).

Also, Vladimir's patch has better time for iterating Hash (hash_flatten, hash_keys and hash_values benchmarks).

#42 - 03/07/2016 05:55 AM - funny_falcon (Yura Sokolov)

Benchmark certainly misses creation of many-many small hashes, where packed representation shines:

```
$ # Trunk
$ /usr/bin/time -v ~/tmp/ruby-trunk/bin/ruby -e '1000000.times.map{|i| a=[];4.times{|j| a[j]=j}};a'

User time (seconds): 1.38
System time (seconds): 0.10
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.49
Maximum resident set size (kbytes): 272572

$ # My patch
$ /usr/bin/time -v ~/tmp/ruby-st/bin/ruby -e '1000000.times.map{|i| a=[];4.times{|j| a[j]=j}};a'

User time (seconds): 1.64
System time (seconds): 0.18
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.83
Maximum resident set size (kbytes): 355452

$ # Vladimir's patch
$ /usr/bin/time -v ~/tmp/ruby-mak/bin/ruby -e '1000000.times.map{|i| a=[];4.times{|j| a[j]=j}};a'

User time (seconds): 1.77
System time (seconds): 0.25
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.02
```

05/08/2021 27/73
We both loose to trunk.

#43 - 03/07/2016 06:02 AM - ko1 (Koichi Sasada)
On 2016/03/07 14:42, funny.falcon@gmail.com wrote:

```
make benchmark-each ITEM=bm_hash OPTS='-r 3 -v' COMPARE_RUBY='<trunk ruby>' is broken :-(
```

That is true. Current rule (benchmark) lead to misunderstanding.

I recommend you to use benchmark/driver.rb script with -e options.

```
ruby benchmark/driver.rb -e trunk::[installed trunk bin] -e
modified::
```

You can also specify miniruby for both versions.

```
make benchmark rule is wrote to check YARV performance easily.
```

--
// SASADA Koichi at atdot dot net

#44 - 03/07/2016 06:35 AM - funny_falcon (Yura Sokolov)
Vladimir, you may borrow ideas from my patch.

#45 - 03/07/2016 07:51 AM - ko1 (Koichi Sasada)
On 2016/03/07 14:17, shyouhei@ruby-lang.org wrote:

```
(WIP is definitely OK with me.) Let me think of a more realistic use case.
```

I believe this is not frequent case. But Ruby is used by many people. We need to care about the Shyouhei's example (if we can).

```
here is random version of deletion and addition test (not only last element) (ref [ruby-core:74182]).

require 'objspace'

h = 10.times.inject({}){|r, i| r[i] = i; r}
100_000.times{|i|
  key = rand(10)
  h.delete(key)
  # add
  h[key] = key
  puts "#{i}	#{ObjectSpace.memsize_of(h)}" if (i % 1_000) == 0
}
```

result:
```
http://www.atdot.net/fp_store/f.jvrn3o/file.copipa-temp-image.png
```

--
// SASADA Koichi at atdot dot net

#46 - 03/07/2016 07:44 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

```
Vladimir, you may borrow ideas from my patch.
```

Thank you, Yura. I definitely investigate them and possibility of their inclusion in the final patch.

```
Still the community should decide about 32-bit hashes and indexes because it is an important part of your patch and probably a big reason for your table size and speed improvements.
```

Here are the results on 4.2GHz i7-4970K for 64- and 32-bit indexes and hashes on your branch. I've calculated speedup by an awk script (as benchmark-each is broken):

```
result:
http://www.atdot.net/fp_store/f.jvrn3o/file.copipa-temp-image.png
```
My opinion is the same -- we should stay with 64-bit for a perspective in the future and fix slowness of work with huge tables if it is possible.

You showed many times that most of languages permit maximum $2^{31}$ elements but probably there are exclusions.

I know only one besides MRI -- Dino (http://dino-lang.github.io). I use it to do my research on performance of dynamic languages. Here are examples to work with huge tables on an Intel machine with 128GB memory:

bash-4.3$ /usr/bin/time ./dino -c 'var t = tab[ ];for (var i = 0;i < 100_000_000; i++) t[i]=i;'  
11.60user 3.10system 0:14.70elapsed 100%CPU (0avgtext+0avgdata 7788892maxresident)k  
0inputs+0outputs (0major+2549760minor)pagefaults 0swaps

Dino uses worse implementation of hash tables than the proposed tables (I wanted to rewrite it for a long time). As I wrote MRI took 7min on the same test for 100_000_000 elements on the same machine. So I guess it is possible to make MRI faster to work with huge hash tables too.

Here is an example with about $2^{30}$ elements:

bash-4.3$ /usr/bin/time ./dino -c 'var t = tab[ ];for (var i = 0;i < 1_000_000_000; i++) t[i]=i;'  
113.97user 60.58system 2:54.57elapsed 99%CPU (0avgtext+0avgdata 89713628maxresident)k  
0inputs+0outputs (0major+39319119minor)pagefaults 0swaps

128GB is not enough for $2^{31}$ elements but I believe 256GB will be enough.

It is still possible to work with about $2^{31}$ elements on 128GB machine if we create table from a vector without several rebuilding as in the above tests:

bash-4.3$ /usr/bin/time ./dino -c 'var v=[2_000_000_000:1],t=tab(v),s=0,i; for (i in t)s+=i;putln(s);'
1999999999000000000

78.02user 38.70system 1:56.85elapsed 99%CPU (0avgtext+0avgdata 96957920maxresident)k

By the way, a value in Dino (16B) is 2 times bigger than in MRI (8B).
On 2016/03/07 14:17, shyouhei@ruby-lang.org wrote:

(WIP is definitely OK with me.) Let me think of a more realistic use case.

I believe this is not frequent case. But Ruby is used by many people. We need to care about the Shyouhei's example (if we can).

here is random version of deletion and addition test (not only last element) (ref [ruby-core:74182]).

```ruby
require 'objspace'

h = 10.times.inject({}){|r, i| r[i] = i; r}
key = rand(10)
h.delete(key)
# add
h[key] = key
puts "#{i}	t#{ObjectSpace.memsize_of(h)}" if (i % 1_000) == 0
```

result:
http://www.atdot.net/fp_store/f.jvrn3o/file.copipa-temp-image.png

Thank you for the test, especially for the graph (its preparation takes some time -- I appreciate it).

I am working on the patch for the table compaction. The patch will be on my branch this evening. The current results for your test look the following.

Trunk:

<table>
<thead>
<tr>
<th></th>
<th>Trunk</th>
<th>1k</th>
<th>2k</th>
<th>3k</th>
<th>4k</th>
<th>5k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>696</td>
<td>696</td>
<td>696</td>
<td>696</td>
<td>696</td>
<td>696</td>
</tr>
</tbody>
</table>

The proposed tables with the patch:

<table>
<thead>
<tr>
<th></th>
<th>Proposed</th>
<th>1k</th>
<th>2k</th>
<th>3k</th>
<th>4k</th>
<th>5k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>752</td>
<td>752</td>
<td>752</td>
<td>752</td>
<td>752</td>
<td>752</td>
</tr>
</tbody>
</table>

The patch keeps the size the same.

Here are the results on 4.2GHz i7-4970K for 64- and 32-bit indexes and hashes on your branch. I've calculated speedup by an awk script (as benchmark-each is broken):

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>32</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_aref_dsym</td>
<td>0.256</td>
<td>0.218</td>
<td>1.17431</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>2.887</td>
<td>2.765</td>
<td>1.04412</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>0.267</td>
<td>0.232</td>
<td>1.15086</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>0.073</td>
<td>0.045</td>
<td>1.62222</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>0.355</td>
<td>0.315</td>
<td>1.12698</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>0.327</td>
<td>0.291</td>
<td>1.12371</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>0.250</td>
<td>0.209</td>
<td>1.19617</td>
</tr>
</tbody>
</table>
After learning that make benchmark is broken and I measured ruby (with 64-bit hashes/indexes) with miniruby (with 32-bit indexes).

Using the advised script benchmark/driver.rb which compares ruby & ruby, I have the following results for Yura's branch:

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_aref_dsym</td>
<td>1.017</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.039</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>1.028</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.035</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>1.007</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.024</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.014</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.013</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.054</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>1.026</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>1.027</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>1.006</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>1.000</td>
</tr>
<tr>
<td>hash_keys</td>
<td>1.177</td>
</tr>
<tr>
<td>hash_shift</td>
<td>0.998</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.047</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.020</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.015</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>1.004</td>
</tr>
<tr>
<td>hash_values</td>
<td>1.090</td>
</tr>
</tbody>
</table>

So the differences are smaller than I posted previously. Sorry for the previous misleading numbers.

#50 - 03/08/2016 02:16 AM - duerst (Martin Dürst)

Vladimir Makarov wrote:

> Still the community should decide about 32-bit hashes and indexes because it is an important part of your patch and probably a big reason for your table size and speed improvements.

There is a Ruby committers' meeting in Tokyo next week. It's not a place for final decisions (but Matz attends, and may make some decisions), but we will be able to discuss your work and in particular the question of 32-bit vs. 64-bit indices. (see https://bugs.ruby-lang.org/projects/ruby/wiki/DevelopersMeeting20160316Japan)

#51 - 03/08/2016 03:49 AM - funny_falcon (Yura Sokolov)

I think, if your intention were to reduce table size when elements counts decreased, then you have error here: https://github.com/vnmakarov/ruby/blob/1ec2199374015033277e7ed82e1a469df73f5b09/st.c#L506-L508

What if there were deleted elements in a middle? Then you should search for current element!
You workaround is still mistaken https://github.com/vnmakarov/ruby/blob/1ec2199374015033277e7ed82e1a469df73f5b09/st.c#L968

About 32bit - it is about cache locality and memory consumption. Million men will pay for 64bit hash and indices, but only three men will ever use this ability. One of this three men will be you showing: "looks, my hash may store > 2^32 Int=>Int pairs (if you have at least 256GB memory)"
Second one will be a man that will check you, and he will say: "Hey, Java can't do it, but Ruby can! (if you have at least 256GB)"
And only third one will ever use it for something useful.
But every one will pay for.

#52 - 03/08/2016 04:01 PM - vmakarov (Vladimir Makarov)

05/08/2021
Yura Sokolov wrote:

I think, if your intention were to reduce table size when elements counts decreased, then you have error here:
https://github.com/vnmakarov/ruby/blob/1ec2199374015033277e7ed82e1a469df73f5b09/st.c#L506-L508

I am aware of it therefore I put ??? in comments. I still think about strategy in changing table sizes. My current variant is far from the final. Actually I wanted to publish this work in April but some circumstances forced me to do it earlier. I am glad that I did it earlier as I am having a great response and now have time to think about issues I was not aware. That is what happens when you publish your work. In GCC community practically all patches are publicly reviewed. I am used to it.

Also I am going still to work on new table improvements which were never discussed here.

What if there were deleted elements in a middle? Then you should search for current element!
You workaround is still mistaken
https://github.com/vnmakarov/ruby/blob/1ec2199374015033277e7ed82e1a469df73f5b09/st.c#L968

OK, thanks for pointing this out. The yesterday patch is buggy. I'll work on fixing it.

About 32bit - it is about cache locality and memory consumption.

32-bit is important for your implementation because it permits to decrease element size from 48B to 32B.
For my implementation, the size of elements will still be the same 24B after switching to 32-bit. Although it permits to decrease overall memory allocated to tables by 20% by decreasing size of the array entries.
I don't like lists (through pointer or indexes). This is a disperse data structure hurting locality and performance on modern CPUs for most frequently used access patterns. The lists were cool long ago when a gap between memory and CPU speed was small.

GCC has analogous problem with RTL on which all back back-end optimizations are done. RTL was designed > 25 years ago and it is a list structure. It is a common opinion in GCC community that it hurts the compiler speed a lot. But GCC community can not change it (it tried several times) because it is everywhere even in target descriptions. One of my major motivation to start this hash table work was exactly in removing the lists.

"Million men will pay for 64bit hash and indices, but only three men will ever use this ability.
One of this three men will be you showing: "looks, my hash may store > 2^{32} Int=>Int pairs (if you have at least 256GB memory)"

I would not underestimate the power of such argument to draw more people to use Ruby.

Especially when it seems that python has not 2^{31} constraint and CPython can create about 2^{30} size dictionary on 128GB machine for less than 4 min.
That is unfortunate that MRI needs 7min for creation of dictionary of 10 times less. I believed it should be fixed although I have no idea how.

Second one will be a man that will check you, and he will say:
"Hey, Java can't do it, but Ruby can! (if you have at least 256GB)"
And only third one will ever use it for something useful.
But every one will pay for.

Although I am repeating myself, the memory size and price can be changed in the near future.

In any case you did your point about hash and table sizes and now it will be discussed on a Ruby's committers meeting.

#53 - 03/08/2016 05:39 PM - funny_falcon (Yura Sokolov)

I don't like lists (through pointer or indexes). This is a disperse data structure hurting locality and performance on modern CPUs for most frequently used access patterns. The lists were cool long ago when a gap between memory and CPU speed was small.

But you destroy cache locality by secondary hash and not storing hashsum in a entries array.

Assume 10000 element hash, lets count cache misses:

My hash:

- hit without collision -- lookup position in bins +1 -- check st_table_entry +1 -- got 2
- hit after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- check second entry +1 -- got 3
- miss with empty bin
- lookup position in bins +1
• got 1
• miss after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- got 2
• miss after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- check second entry +1 -- got 3

Your hash:
• hit without collision -- lookup position in entries +1 -- check st_table_element +1 -- got 2
• hit after one collision -- lookup position in entries +1 -- check st_table_entry +1 -- lookup second position in entries +1 -- check second element +1 -- got 4
• miss with empty entry
• lookup position in entries +1
• got 1
• miss after one collision -- lookup position in entries +1 -- check st_table_element +1 -- check second position in entries +1 -- got 3
• miss after one collision -- lookup position in entries +1 -- check st_table_element +1 -- check second position in entries +1 -- check second entry +1 -- check third position in entries +1 -- got 5

So your implementation always generates more cache misses than mine. You completely destroy whole idea of open-addressing.

To overcome this issue you ought use fill factor 0.5. Providing, you don't use 32bit indices, you spend at least 24+8*2=40 bytes per element - just before rebuilding. And just after rebuilding entries with table you spend 24*2+8*2=80bytes per element!
That is why your implementation doesn't provide memory saving either.

My current implementation uses at least 32+4/1.5=34 bytes, and at most 32*1.5+4=52 bytes. And I'm looking for possibility to not allocate double-linked list until necessary, so it will be at most 24*1.5+4=40 bytes for most of hashes.

Lists are slow when every element is allocated separately. Then there is also TLB miss together with cache miss for every element. When element are allocated from array per hash, then there are less both cache and TLB misses.

And I repeat again: you do not understand when and why open addressing may save cache misses. For open addressing to be effective one need to store whole thing that needed to check hit in an array itself (so at least hash sum owt to be stored). And second probe should be in a same cache line, that limits you:
• to simple schemes: linear probing, quadratic probing,
• or to custom schemes, when you explicitely check neighbours before long jump,
• or exotic schemes, like Robin-Hood hashing.

You just break every best-practice of open-addressing.

#54 - 03/08/2016 08:15 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

I don't like lists (through pointer or indexes). This is a disperse data structure hurting locality and performance on modern CPUs for most frequently used access patterns. The lists were cool long ago when a gap between memory and CPU speed was small.

But you destroy cache locality by secondary hash and not storing hashsum in a entries array.

Assume 10000 element hash, lets count cache misses:

My hash:
• hit without collision -- lookup position in bins +1 -- check st_table_entry +1 -- got 2
• hit after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- check second entry +1 -- got 3
• miss with empty bin
• lookup position in bins +1
• got 1
• miss after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- got 2
• miss after one collision -- lookup position in bins +1 -- check st_table_entry +1 -- check second entry +1 -- got 3

Your hash:
• hit without collision -- lookup position in entries +1 -- check st_table_element +1 -- got 2
• hit after one collision -- lookup position in entries +1 -- check st_table_entry +1 -- lookup second position in entries +1 -- check second element +1 -- got 4
• miss with empty entry
• lookup position in entries +1
• got 1
• miss after one collision -- lookup position in entries +1 -- check st_table_element +1 -- check second position in entries +1 -- got 3
• miss after one collision -- lookup position in entries +1 -- check st_table_element +1 -- check second position in entries +1 -- check second entry +1 -- check third position in entries +1 -- got 5

So your implementation always generates more cache misses than mine. You completely destroy whole idea of open-addressing.
What is missing in the above calculations is the probability of collisions for the same size table. The result can be not so obvious (e.g. in first case we have collision in 20% but in the second one only 10%): 2 + 3/5 vs 2 + 4/10 or 2.6 vs 2.4 cache misses.

But I am writing about it below.

To overcome this issue you ought use fill factor 0.5.

Providing, you don’t use 32bit indices, you spend at least 24+8*2=40 bytes per element - just before rebuilding.

And just after rebuilding entries with table you spend 24*2+8*2=80 bytes per element.

That is why your implementation doesn’t provide memory saving either.

One test mentioned in this thread showed that in 3 cases out of 4 my tables are more compact than the current one.

My current implementation uses at least 32+4/1.5=34 bytes, and at most 32*1.5+4=52 bytes.

And I’m looking for possibility to not allocate double-linked list until necessary, so it will be at most 24*1.5+4=40 bytes for most of hashes.

It would be a fair comparison if you used 64-bit vs 64-bit. As I wrote 32-bit hashes and indexes are important for your implementation. So for 64-bit the numbers should look like

at least 48 + 8 / 1.5 = 51 bytes (vs. 40 for my approach)

and what number of collisions would be in the above case. Pretty big if you are planning in average 1.5 elements for a bin.

at most 48 * 1.5 + 8 = 80 bytes (vs. 80)

analogously a big number of collisions if you plan in average 1 element for a bin

If the community decides that we should constrain the table sizes, then I might reconsider my design. When I started work on the tables, I assumed that I can not put additional constraints to existing tables in other words that I should not change the sizes. To be honest at the start I also thought what if the index were 32-bit (although I did not think further about changing hash size also as you did).

Lists are slow when every element is allocated separately. Then there is also TLB miss together with cache miss for every element.

When element are allocated from array per hash, then there are less both cache and TLB misses.

It is still the same cache miss when the next element in the bin list is out of cache line and it is very probable if you have moderate or big hash tables.

It is an improvement that you put elements in the array, at least they will be closer to each other and will be not some freed elements created for other tables. Still, IMHO, it is practically the same pointer chasing problem.

And I repeat again: you do not understand when and why open addressing may save cache misses.

I think I do understand “why open addressing may save cache misses”.

My understanding is that it permits to remove the bin (bucket) lists and decreases size of the element. As a consequence you can increase the array entries and have a very healthy load factor practically excluding collision occurrences while having the same size for the table.

For open addressing to be effective one need to store whole thing that needed to check hit in an array itself (so at least hash sum owt to be stored).

That will increase the entry size. It means that for the same size table I will have a bigger load factor. Such solution decreases cache misses in case of collisions but increases the collision probability for the same size tables.

And second probe should be in a same cache line, that limits you:

- to simple schemes: linear probing, quadratic probing,

As I wrote only one thing interesting for me in quadratic probing is better data locality in case of collisions, I'll try it and consider if my final patch will be in the trunk.

- or to custom schemes, when you explicitely check neighbours before long jump,
- or exotic schemes, like Robin-Hood hashing.

You just break every best-practice of open-addressing.

No, I don’t. IMHO.

You also omitted table traverse operations here. I believe in my approach it will work better. But I guess we could argue also about it a lot.
When the meeting decides about the sizes, we will have more clarity. There is no ideal solutions and speculations sometimes might be interesting but the final results on benchmarks should be a major criterion. Although our discussions are sometimes emotional, the competition will help to improve Ruby hash tables which is good for MRI community.

#55 - 03/08/2016 08:29 PM - vmakarov (Vladimir Makarov)

Vladimir Makarov wrote:

What is missing in the above calculations is the probability of collisions for the same size table. The result can be not so obvious (e.g., in first case we have collision in 20% but in the second one only 10%): 2 + 3/5 vs 2 + 4/10 or 2.6 vs 2.4 cache misses.

Sorry, I made a mistake in calculations for given conditions. It should be 2*0.8 + 3*0.2 vs 2*0.9 + 4*0.1 or 2.2 vs 2.2. It is just an illustration of my thesis. The different collision probabilities can change the balance to different directions.

#56 - 03/15/2016 01:30 PM - funny_falcon (Yura Sokolov)

- File 0001-st.c-change-st_table-implementation.patch added

Good day, everyone.

I'm presenting my (pre)final version of patch. It passes all tests and redmine works on it without issues.

Vladimir, I must admit your original idea about rebuilding st_table_entries were correct, so there is no need in double-linked list. Ruby is not for huge hashes in low latency applications, and GC still ought to traverse whole hash with marking referenced objects.

Differences from Vladimir's approach:

- closed addressing, with fill factor close to 1, (so no need to keep counter for deleted entries and rebuild main hash array).
- 32bit hashes and indices on 64bit platform
- size is stored as index to static table, so allocation is not always 2x, but instead it is 1.5x and 1.33x
- single allocation both for entries and bins, pointer points into middle of allocation, bins array is on one side, and entries on other.

Patch url https://github.com/ruby/ruby/compare/trunk...funny-falcon:st_table_with_array2.patch
Also attached to ticket.

Results for Redmine.
Unfortunately, Redmine works with mistake under current Vladimir's branch, so there is no results for -:

There is 2-3% performance improvement.
When configured with --without-jemalloc, patched version uses 12% less memory.
When configured with --with-jemalloc, only 5% less memory used.

(webserver is puma, database postresql, 5 issues in a list, 1 issue with 6 comments)

$ # trunk without jemalloc
$ ab -n 1000 -c 10 http://localhost:3000/projects/general/issues
Requests per second: 23.88 [#/sec] (mean)
$ ab -n 1000 -c 10 http://localhost:3000/issues/1
Requests per second: 24.33 [#/sec] (mean)
$ ps aux | grep puma
yura 17709  60.6 1.4 1453416 242636 pts/31 Sl+ 15:44 1:33 puma 3.1.0 (tcp://0.0.0.0:3000) [redmine-trunk]

$ # trunk + jemalloc
$ ab -n 1000 -c 10 http://localhost:3000/projects/general/issues
Requests per second: 28.25 [#/sec] (mean)
$ ab -n 1000 -c 10 http://localhost:3000/issues/1
Requests per second: 28.27 [#/sec] (mean)
$ ps aux | grep puma
yura 32297  69.9 1.7 490356 285568 pts/31 Sl+ 15:09 2:23 puma 3.1.0 (tcp://0.0.0.0:3000) [redmine-trunk]

$ # patch without jemalloc
$ ab -n 1000 -c 10 http://localhost:3000/projects/general/issues
Requests per second: 24.60 [#/sec] (mean)
$ ab -n 1000 -c 10 http://localhost:3000/issues/1
Requests per second: 24.82 [#/sec] (mean)
$ ps aux | grep puma
yura 29489  60.5 1.3 1378392 213956 pts/34 Sl+ 15:52 2:02 puma 3.1.0 (tcp://0.0.0.0:3000) [redmine-st]

$ # patch + jemalloc

05/08/2021
Good day, everyone.

I'm presenting my (pre)final version of patch.

Thanks. I'll investigate your patch later.

Here is my work update. This is far from the final version. I'll continue to work on it. As I wrote I can spend more time on MRI work after mid-April (currently I'd like to inspect and optimize my code and investigate more about hash functions and tune them).

I am glad that I submitted my patch earlier than I planned originally. The discussion was useful for me.

The discussion reveals that I used a wrong benchmarking procedure (I took it reading emails on ruby developer mailing lists):

```
make -benchmark-each ITEM=bm_hash OPTS=-r 3 -v' COMPARE_RUBY=
```

This command measures installed ruby with the current miniruby. So the results reported by me were much better than in reality. Thanks to Koichi Sasada, now I am using the right one:

```
ruby ../gruby/benchmark/driver.rb -p hash -r 3 -e  -e
```

So I realized that I should work more to improve the performance results because average 15% improvement is far away than 50% reported the first time.

Since the first patch submitting, I did the following (I still use 64-bit long hashes and indexes):

- I changed terminology to keep the same API. What I called elements is now called entries and what I called entries are now called bins.
- I added a code for table consistency checking which helps to debug the code (at least it helped me a lot to find a few bugs).
- I implemented the compaction of the array entries and fixed a strategy for table size change. It fixes the reported memory leak.
- I made the entries array is cyclical to exclude overhead of table compaction or/and table size change for usage the hash tables as a queue.
- I implemented a compact representation of small tables of up to 8 elements. I also added tests for small tables of size 2, 4, 8 to check small hash tables performance.
- I also tried to place hashes inside bins to improve data locality in cases of collisions as Yura wrote. It did not help. The average results were a bit worse. I used the same number of elements in the bins. So the array bins became 2 times bigger and probably that...
worsened the locality. I guess tuning ratio $\#\text{bin elements}/\#\text{entries}$ could improve the results but I believe the improvement will not worth of doing this. Also implementing a better hashing will probably make the improvement impossible at all.

- Working on the experiment described above I figured that sometimes MRI hash functions produces terrible hashes and the collisions achieves 100% on some benchmarks. This is bad for open addressing hash tables where big number of collisions results in more cache line reads than for tables with chains. Yura Sokolov already wrote about this.

  - I ran ruby murmur and sip24 hashing on smhasher test ([https://github.com/aappleby/smhasher](https://github.com/aappleby/smhasher)). MurMurHashing in st.c in about 3 faster than sip24 on bulk speed test (2GB/sec to 6GB/s) but murmur purely performs practically on all hashing quality tests except differential Tests and Keyset ‘Cyclic’ Tests. E.g. on avalanche tests the worst bias is 100% (vs 0.6% for sip24). It is very strange because all murmur hash functions from smhasher behave well.
  - I did not start to figure out the reason because I believe we should use City64 hash (distributed under MIT license) as currently the fastest high quality non-crypto hash function. Its speed achieves 12GB/s. So I changed murmur by City64.
  - I also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests. It permits to improve collisions for these tests, e.g. from 73% to 0.3% for hash_ident_num.
  - I believe usage of City64 will help to improve the table performance for most widely used case when the keys are strings.

- Examining siphash24. I got to conclusion that although using a fast crypto-level hash function is a good thing but there is a simpler solution to solve the problem of possible denial attack based on hash collisions.

  - When a new table element is inserted we just need to count collisions with entries with the same hash (more accurately a part of the hash) but different keys and when some threshold is achieved, we rebuild the table and start to use a crypto-level hash function. In reality such function will be never used until someone really tries to do the denial attack.
  - Such approach permits to use faster non-crypto level hash functions in the majority of cases. It also permits easily to switch to usage of other slower crypto-level functions (without losing speed in real world scenario), e.g. sha-2 or sha-3. Siphash24 is pretty a new functions and it is not time-tested yet as older functions.
  - So I implemented this approach.

- I also tried double probing as Yora proposed. Although performance of some benchmarks looks better it makes worse code in average (average performance decrease is about 1% but geometric mean is about 14% because of huge degradation of hash_aref_flo). I guess it means that usage of the double probing can produce better results because of better data locality but also more collisions for small tables as it always uses only small portions of hashes, e.g. 5-6 lower bits. It might also mean the the specialized ruby hash function still behaves purely on flonums although all 64 bits of the hash avoids collisions well. Some hybrid scheme of usage of double probing for big tables and secondary hash function using other hash bits for small tables might improve performance more. But I am a bit skeptical about such scheme because of additional overhead code.

All the described work permitted to achieve about 35% and 53% (using the right measurements) better average performance than the trunk correspondingly on x86-64 (Intel i7-4790K) and ARM (Exynos 5410).

I've just submitted my changes to the github branch (again it is far from the final version of the patch). The current version of the patch can be seen as

[https://github.com/vnmakarov/ruby/compare/trunk...hash_tables_with_open_addressing.patch](https://github.com/vnmakarov/ruby/compare/trunk...hash_tables_with_open_addressing.patch)

**#60 - 03/15/2016 06:23 PM - funny_falcon (Yura Sokolov)**

Murmur is not used for Hash, cause it is target for hashDoS - it has seed independent collisions. City64 also has seed independent collisions. That is why SipHash were born and adopted by community.


But SipHash could be relaxed:

- currently it is SipHash24 - 2 rounds for 8byte block and 4 rounds for finalization
- but even SipHash author confirms that for internal hash table (ie when hash sum is not exposed to attacker) SipHash13 is just enough.

Single call to st_hash in hash.c is just to combine two hashsums (calculated with SipHash) into one.

All usage of st_init_strtable are not performance critical.

I made the entries array is cyclical to exclude overhead of table compaction or/and table size change for usage the hash tables as a queue.

I doubt using hash table as a queue is a useful case. But I could be mistaken. And cyclic allocation doesn't solve LRU use case at all! :-(

also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests

I've seen you do it. Great catch! I also fixed it in other way (cause I don't use perturb I must be ensure all bits are mixed into lower bits).

I also tried double probing.

It takes me a time to realize that you mean quadratic probing :-(

double probing may be: test slot, then test neighbor, do long jump and repeat. It is not wildly used technique.

Do you configure with --with-jemalloc?

trunk is much faster with jemalloc linked, and it is hard to beat it by performance.

Unfortunately, Redmine still doesn't work with your branch, so I can't benchmark it.

#61 - 03/15/2016 06:26 PM - funny_falcon (Yura Sokolov)

I made the entries array is cyclical to exclude overhead of table compaction or/and table size change for usage the hash tables as a queue.

I doubt using hash table as a queue is a useful case. But I could be mistaken. And cyclic allocation doesn't solve LRU use case at all! :-(

I mean, I really think st_table_entry compaction is a right way to go (so I agree with your initial proposal in this area).

In my implementation I just check that there is at least 1/4 of entries are deleted. If so, then I compact table, else grow.

#62 - 03/15/2016 07:37 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

Murmur is not used for Hash, cause it is target for hashDoS - it has seed independent collisions.

I did no write that murmur is used for Hash. It is still used internally.

City64 also has seed independent collisions.

City64 is 2 faster than murmur so it is better to use it where murmur is used. Ruby murmur code is also broken (bad hash quality) but looking at your changes in murmur MRI code you probably fixed it.

That is why SipHash were born and adopted by community.

http://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/

But SipHash could be relaxed:

• currently it is SipHash24 - 2 rounds for 8byte block and 4 rounds for finalization
• but even SipHash author confirms that for internal hash table (ie when hash sum is not exposed to attacker) SipHash13 is just enough. https://github.com/rust-lang/rust/issues/29754#issuecomment-156073946

Single call to st_hash in hash.c is just to combine two hashsums (calculated with SipHash) into one.
All usage of st_init_strtable are not performance critical.

Yura, I suspect you missed my point. What I proposed is to use also City24 in cases where siphash24 is currently used and use siphash24 only when the attack is on the way. Therefore I added a code to the tables to recognize the attack and switch to crypto-level hash function which in reality will be extremely rare. City24 is 6 times faster than siphash24, therefore Hash will be faster and still crypto-strong. Moreover you can use other crypto-level hash functions without losing the speed for real world scenarios.

I made the entries array is cyclical to exclude overhead of table compaction or/and table size change for usage the hash tables as a queue.

I doubt using hash table as a queue is a useful case. But I could be mistaken. And cyclic allocation doesn't solve LRU usecase at all :-(

Still it is nice to have because it costs practically nothing (one additional mask operation whose tiny latency time will be hidden between execution of parallelly executed insns on modern superscalar OOO processors).

also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests

I've seen you do it. Great catch!
I also fixed it in other way (cause I don't use perturb I must be ensure all bits are mixed into lower bits).

I also tried double probing

It takes me a time to realize that you mean quadratic probing :-) double probing may be: test slot, then test neighbor, do long jump and repeat. It is not wildly used technique.

Sorry, my memory failed me again. That was quadratic probing which you proposed to use for open-addressing tables.

Do you configure with --with-jemalloc ? trunk is much faster with jemalloc linked, and it is hard to beat it by performance.

Not yet but will do.

Unfortunately, Redmine still doesn't work with your branch, so I can't benchmark it.

Thanks. I'll investigate this. Unfortunately I am too far away from web development and have no experience with such applications but I will learn it.

The most exciting thing for me is that better hash tables can improve the performance of real applications. Before this, I looked at the table work more as an exercise to become familiar with MRI before deciding to do a real performance work.

#63 - 03/15/2016 08:13 PM - funny_falcon (Yura Sokolov)

The most exciting thing for me is that better hash tables can improve the performance of real applications.

So then you should benchmark with real application.

For example, you will be surprised how small effect gives changing siphash to faster function.

By the way, do you run make check regulary?

#64 - 03/15/2016 08:58 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

The most exciting thing for me is that better hash tables can improve the performance of real applications.

So then you should benchmark with real application.

For example, you will be surprised how small effect gives changing siphash to faster function.
By the way, do you run make check regularly?

Not really. I do regularly make test. I did make check on my branch point before committing my first patch to the branch but I had a lot of messages something about leaked file descriptor. So I thought make check was broken.

I guess I should not ignore it. Just running it now I see a failure on test_numhash.rb. I am just going to start work on fixing it.

Thanks for mentioning it.

---

#65 - 03/16/2016 12:08 AM - normalperson (Eric Wong)

vmakarov@redhat.com wrote:

- I also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests. It permits to improve collisions for these tests, e.g. from 73% to 0.3% for hash_ident_num.

Nice. Do you think it's worth it to split this change out for use with the current st implementation?

---

#66 - 03/16/2016 05:32 AM - funny_falcon (Yura Sokolov)

Eric Wong wrote:

vmakarov@redhat.com wrote:

- I also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests. It permits to improve collisions for these tests, e.g. from 73% to 0.3% for hash_ident_num.

Nice. Do you think it's worth it to split this change out for use with the current st implementation?

Eric, Vladimir’s change considers it's use of high bits inside of hash table implementation.

You may take the same function from my patch: cause my hash table uses same close addressing as current trunk st, I ought to mix high bits into lower.

Also, there is a fix for current murmur_finish function.

---

#67 - 03/16/2016 02:57 PM - vmakarov (Vladimir Makarov)

Eric Wong wrote:

vmakarov@redhat.com wrote:

- I also changed the specialized hash function (rb_num_hash_start) used for bm_hash_ident tests. It permits to improve collisions for these tests, e.g. from 73% to 0.3% for hash_ident_num.

Thank you. Right now it has a little sense especially for small tables. The current implementation simply ignores a lot of higher bits, e.g. a table with 32-bit entries uses only 5 lower bits of 64-bit hash. The change I've done is good when you use all 64-bits of the hash and my implementation finally consumes all these 64 bits although several collisions should occur for this.

To be a good hash function when you use only small lower part of the hash, the function should behave well on avalanche test. It means that when you change just one bit in any part of the key, "avalanche" of changes happens in the result hash (the best when half of hash bits through the all hash changes). I guess any simple specialized function will be bad on such test. Major their advantage is high speed. Murmur, City64, SipHash24 are quite good on such tests but they are much slower.

I think I'll have a few patches when I am done with the hash tables: the hash table itself, hash functions, code for recognizing a denial attack and switching to stronger hash functions. I am not sure that all (or any) will be finally accepted.

---

#68 - 03/17/2016 06:00 AM - shyouhei (Shyouhei Urabe)

About 64bit versus 32bit index: several developers discussed this on this month's developer meeting. Consensus there was that we do not want to limit Hash size, while it is true that over 99% of hashes are smaller than 4G entries, and it is definitely a good idea to optimize to them. We did not reach a consensus as to how should we achieve that, though. Proposed ideas include: switch index type using configure, have 8/16/32/64bit versions in parallel and switch smoothly with increasing hash size.

---

#69 - 03/17/2016 07:51 AM - ko1 (Koichi Sasada)
On 2016/03/17 15:00, shyouhei@ruby-lang.org wrote:

About 64bit versus 32bit index: several developers discussed this on this month's developer meeting. Consensus there was that we do not want to limit Hash size, while it is true that over 99% of hashes are smaller than 4G entries, and it is definitely a good idea to optimize to them. We did not reach a consensus as to how should we achieve that, though. Proposed ideas include: switch index type using configure, have 8/16/32/64bit versions in parallel and switch smoothly with increasing hash size.

Compliment:

(1) switch index type using configure

Easy. However, we can not test both versions.

(2) have 8/16/32/64bit versions in parallel and switch smoothly with increasing hash size.

There is one more:

(3) st only supports 32bit, but Hash object support 64bit using another technique (maybe having multiple tables).

Anyway, we concluded that this optimization should be introduced for Rub 2.4 even if it only supports 64bit. Optimization with 32bit length index is nice, but we can optimize later.

--
// SASADA Koichi at atdot dot net

#70 - 03/17/2016 09:03 AM - duerst (Martin Dürst)

On 2016/03/17 16:50, SASADA Koichi wrote:

On 2016/03/17 15:00, shyouhei@ruby-lang.org wrote:

About 64bit versus 32bit index: several developers discussed this on this month's developer meeting. Consensus there was that we do not want to limit Hash size, while it is true that over 99% of hashes are smaller than 4G entries, and it is definitely a good idea to optimize to them. We did not reach a consensus as to how should we achieve that, though.

Matz also at the beginning of the discussion of this issue said that currently, he would emphasize speed over size, so if e.g. the 32-bit version led to a 10% speed increase, going with 32-bit would be okay.

We also agreed that currently, a Hash with more than 4G elements is mostly theoretical, but that will change in the future. So it may be okay to stay with 32-bit if we can change it easily to 64-bit in the future.

#71 - 03/17/2016 10:47 AM - funny_falcon (Yura Sokolov)

I strongly prefer configure option with default to 32bit.
99% should not pay for 1% ability.

32bit version doesn't lead to performance improvement (may be 1%, not more).
But it leads to memory consumption reduction (up to 10%).

(3) st only supports 32bit, but Hash object support 64bit using another technique (maybe having multiple tables).

Is it ok for Hash to have different implementation than st_table?
There is source level dependency in some gems for RHASH_TBL :-(
May we force library authors to fix their libraries?

It will be very promising to switch implementation to something that needs one allocation for small hashes instead of current two allocations (one for st_table and one for entries).
Also, implementation could be more tightly bound with Hash needs and usage patterns.

#72 - 03/17/2016 02:33 PM - vmakarov (Vladimir Makarov)

Koichi Sasada wrote:
On 2016/03/17 15:00, shyouhei@ruby-lang.org wrote:

About 64bit versus 32bit index: several developers discussed this on this month's developer meeting. Consensus there was that we do not want to limit Hash size, while it is true that over 99% of hashes are smaller than 4G entries, and it is definitely a good idea to optimize to them. We did not reach a consensus as to how should we achieve that, though. Proposed ideas include: switch index type using configure, have 8/16/32/64bit versions in parallel and switch smoothly with increasing hash size.

Compliment:

1) switch index type using configure

Easy. However, we can not test both versions.

2) have 8/16/32/64bit versions in parallel and switch smoothly with increasing hash size.

There is one more:

3) st only supports 32bit, but Hash object support 64bit using another technique (maybe having multiple tables).

Thank you for the clarification.

I'll investigate the possibility of usage less bits for indexes (and may be hashes) without introducing constraints on the table size. Right now I can say that using 32-bit for 64-bit Haswell improves the average performance of my tables by about 2% on MRI hash benchmarks. On my estimations, the size should be decreased by 12.5%. So it is worth to investigate this.

Anyway, we concluded that this optimization should be introduced for Rub 2.4 even if it only supports 64bit. Optimization with 32bit length index is nice, but we can optimize later.

I'll work on it. I thought that my patch was a final variant of the table when I submitted it first. But having all the feedback I believe I should work more on the code.

As I wrote I'll probably submit a few patches because some work I've already done might seem controversial. I think about the new hash table patch, new hash functions patch, and a patch for the tables dealing with a denial attack. Yura Sokolov could submit his patch for using alternative allocation.

I planning to do some research, actual coding and submit the patches definitely before the summer but it might happen earlier (it depends on how busy I am with my major work). I believe we will have enough time to introduce the new hash tables in 2.4.

#73 - 03/31/2016 07:32 PM - normalperson (Eric Wong)
vmakarov@redhat.com wrote:

I think I'll have a few patches when I am done with the hash tables: the hash table itself, hash functions, code for recognizing a denial attack and switching to stronger hash functions. I am not sure that all (or any) will be finally accepted.

I look forward to these and would also like to introduce some new APIs for performance.

One feature would be the ability to expose and reuse calculated hash values between different tables to reduce hash function overheads.

This can be useful in fstring and symbol tables where the same strings are repeatedly reused as hash keys.

For example, I was disappointed we needed to revert r43870 (git commit 0c3b3e9237e8) in [Misc #9188] which deduplicated all string keys for Hash#[]=:

https://bugs.ruby-lang.org/issues/9188

So, perhaps being able to reuse hash values between different tables can improve performance to the point where we can dedupe all string hash keys to save memory.

I am also holding off on committing a patch to dedupe keys
from Marshal.load [Feature #12017] because I hope to resurrect r43870 before Ruby 2.4:

https://bugs.ruby-lang.org/issues/12017

I had another feature in mind, but can't remember it at the moment :x

I doubt I'll have time to work on any of this until you're done with your work. My time for ruby-core is limited for the next 2-3 months due to vacations and other projects.

#74 - 04/01/2016 03:41 AM - vmakarov (Vladimir Makarov)
- File st-march31.patch added

Eric Wong wrote:

vmakarov@redhat.com wrote:

I think I'll have a few patches when I am done with the hash tables: the hash table itself, hash functions, code for recognizing a denial attack and switching to stronger hash functions. I am not sure that all (or any) will be finally accepted.

I look forward to these and would also like to introduce some new APIs for performance.

Thank you for sharing this. I am still working on the hash tables when I have spare time.

I already implemented variable indexes (8-, 16-, 32-, 64-bits). It gave about 3% average improvement on MRI hash benchmarks. I tried a lot and did some other changes too. __builting_prefetch gave a good improvement while __builtin_expect did nothing. So the current average improvement on MRI hash benchmarks is close to 40% on Intel Haswell and >55% on ARMv7 (using the right comparison of one miniruby vs. another miniruby).

One feature would be the ability to expose and reuse calculated hash values between different tables to reduce hash function overheads.

This can be useful in fstring and symbol tables where the same strings are repeatedly reused as hash keys.

For example, I was disappointed we needed to revert r43870 (git commit 0c3b3e9237e8) in [Misc #9188] which deduplicated all string keys for Hash[]'=:

https://bugs.ruby-lang.org/issues/9188

So, perhaps being able to reuse hash values between different tables can improve performance to the point where we can dedupe all string hash keys to save memory.

The hash reuse would be nice. Hash calculation can be expensive. May be it even will permit to remove hash from the entries at least w/o losing performance. I actually tried to remove hash and recalculate it when it is necessary but it gave worse performance (i also tried to put smaller hashes into bins to decrease cache misses -- it worked better but still worse than storing full hash in entries).

I am also holding off on committing a patch to dedupe keys from Marshal.load [Feature #12017] because I hope to resurrect r43870 before Ruby 2.4:

https://bugs.ruby-lang.org/issues/12017

I had another feature in mind, but can't remember it at the moment :x

I doubt I'll have time to work on any of this until you're done with your work. My time for ruby-core is limited for the next 2-3 months due to vacations and other projects.

This project is holding me back too (it can be quite addicted but I feel I am already repeating the same ideas with small variations). I'd like to move on to another project. So I'll try to submit a major patch before the end of April in order you have time to work on the new code.
I am sending the current overall patch. I think it is pretty close to what I'd like to see finally. I hope the patch will help with your work.

The patch is relative to trunk at 7c14876b2fbbd61bda95e42dea82c954fa9d0182 which was the head about 1 month ago.

#75 - 04/01/2016 07:13 AM - funny_falcon (Yura Sokolov)

average improvement on MRI hash benchmarks is close to 40% on Intel Haswell and >55% on ARMv7

What about realworld applications like Redmine?

#76 - 04/05/2016 10:21 AM - funny_falcon (Yura Sokolov)

- File deleted (0001-st.c-use-array-for-storing-st_table_entry.patch)

#77 - 04/29/2016 10:36 PM - vmakarov (Vladimir Makarov)

- File base.patch added
- File hash.patch added
- File strong_hash.patch added
- File city.patch added

Since submitting my first patch for new Ruby hash tables, a lot of code was reworked and a lot of new implementation details were tried. I feel that I reached a point from where I can not improve the hash table performance anymore. So I am submitting the final patches (of course I am ready to change code if/after people review it and propose meaningful changes).

I broke the code into 4 patches.

The first patch is the base patch. The most work I did for new hash table implementation is in this patch. The patch alone gives 32-48% average performance improvement on Ruby hash table benchmarks

The second patch changes some simple hash functions to make them faster. I gives 5-8% speedup additionally.

The third patch might seem controversial. It implements a code to use faster but not strong hash functions and to recognize hash table denial attacks and switch to stronger hash function (currently to Ruby siphash). It gives additional 6-7% average performance improvement.

The forth patch changes MurMur hash to Google City64 hash. Although on very long keys City64 is two times faster than MurMur in Ruby, its usage actually makes Ruby hash table benchmarks slower. So I do not propose to add this patch to Ruby trunk. I put this patch here only for people who might be interesting to play with it.

All patches were tested on x86/x86-64, ARM, and PPC64 with make check. I did not find any additional test regressions.

The patches were also benchmarked on Ruby hash table benchmarks on Intel 4.2GHz i7-4970K, 2GHz ARM Exynos5422, and 3.55GHz Power7. This time I used more accurate measurements using the following script:

ruby ../ruby/benchmark/driver.rb -p hash -r 3 -e trunk::<trunk-miniruby> -e current::<miniruby-with-patches> 2 >/dev/null|awk 'NF==2 && /hash/ {s+=$2;n++;print} END{print s/n}'

Here are the average performance results of the patches relative to the old hash tables (old execution time / new execution time):

<table>
<thead>
<tr>
<th></th>
<th>x86-64</th>
<th>ARM</th>
<th>PPC64</th>
</tr>
</thead>
<tbody>
<tr>
<td>base patch</td>
<td>1.32326</td>
<td>1.48507</td>
<td>1.36359</td>
</tr>
<tr>
<td>above+hash patch</td>
<td>1.39067</td>
<td>1.53537</td>
<td>1.44748</td>
</tr>
<tr>
<td>above+strong hash patch</td>
<td>1.45615</td>
<td>1.59833</td>
<td>1.50193</td>
</tr>
<tr>
<td>above+City64 patch</td>
<td>1.43611</td>
<td>1.58185</td>
<td>1.48878</td>
</tr>
</tbody>
</table>

If somebody is interesting in the results for each benchmark, I put them at the end of the message.
Are the first three patches OK for the trunk? If it is so, I don't know how to commit them to the trunk. So could somebody commit them or explain the procedure how to do it in this case.

<table>
<thead>
<tr>
<th>base patch:</th>
<th>x86-64</th>
<th>ARM</th>
<th>PPC64</th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.640</td>
<td>1.523</td>
<td>1.062</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>0.929</td>
<td>0.924</td>
<td>0.950</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.419</td>
<td>1.445</td>
<td>1.366</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>0.857</td>
<td>0.885</td>
<td>1.002</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.433</td>
<td>1.078</td>
<td>1.136</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>1.011</td>
<td>0.897</td>
<td>0.989</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>0.975</td>
<td>0.979</td>
<td>0.940</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.002</td>
<td>0.910</td>
<td>0.999</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.035</td>
<td>0.923</td>
<td>1.025</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.184</td>
<td>1.282</td>
<td>1.377</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.930</td>
<td>1.087</td>
<td>1.054</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.911</td>
<td>0.931</td>
<td>0.987</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.950</td>
<td>0.937</td>
<td>0.992</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.950</td>
<td>0.908</td>
<td>1.002</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.936</td>
<td>0.907</td>
<td>0.990</td>
</tr>
<tr>
<td>hash_keys</td>
<td>2.793</td>
<td>1.365</td>
<td>1.786</td>
</tr>
<tr>
<td>hash_long</td>
<td>0.993</td>
<td>0.949</td>
<td>1.008</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.265</td>
<td>2.615</td>
<td>1.735</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.292</td>
<td>2.208</td>
<td>1.724</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.286</td>
<td>2.192</td>
<td>1.672</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.287</td>
<td>2.003</td>
<td>1.651</td>
</tr>
<tr>
<td>hash_small2</td>
<td>0.991</td>
<td>1.018</td>
<td>0.946</td>
</tr>
<tr>
<td>hash_small4</td>
<td>0.993</td>
<td>0.990</td>
<td>0.992</td>
</tr>
<tr>
<td>hash_small8</td>
<td>2.186</td>
<td>2.195</td>
<td>2.301</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>1.036</td>
<td>1.032</td>
<td>1.026</td>
</tr>
<tr>
<td>hash_values</td>
<td>2.781</td>
<td>1.303</td>
<td>1.751</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>2.663</td>
<td>6.611</td>
<td>4.354</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base+hash patches:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.594</td>
<td>1.193</td>
<td>1.126</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>0.942</td>
<td>0.920</td>
<td>0.983</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.421</td>
<td>1.438</td>
<td>1.362</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>1.031</td>
<td>0.961</td>
<td>1.139</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.921</td>
<td>1.093</td>
<td>1.455</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>1.047</td>
<td>1.010</td>
<td>1.058</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.039</td>
<td>0.981</td>
<td>1.030</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.000</td>
<td>0.897</td>
<td>1.000</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.039</td>
<td>0.899</td>
<td>1.041</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.122</td>
<td>1.290</td>
<td>1.326</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.962</td>
<td>0.966</td>
<td>1.091</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.957</td>
<td>0.935</td>
<td>1.035</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.952</td>
<td>0.959</td>
<td>0.990</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.942</td>
<td>0.931</td>
<td>0.976</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.967</td>
<td>0.859</td>
<td>0.990</td>
</tr>
<tr>
<td>hash_keys</td>
<td>2.733</td>
<td>1.364</td>
<td>1.681</td>
</tr>
<tr>
<td>hash_long</td>
<td>1.012</td>
<td>0.970</td>
<td>1.028</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.402</td>
<td>2.973</td>
<td>1.894</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.332</td>
<td>2.367</td>
<td>1.720</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.317</td>
<td>2.374</td>
<td>1.735</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.330</td>
<td>2.169</td>
<td>1.705</td>
</tr>
<tr>
<td>hash_small2</td>
<td>1.010</td>
<td>0.965</td>
<td>0.938</td>
</tr>
<tr>
<td>hash_small4</td>
<td>1.006</td>
<td>0.962</td>
<td>1.013</td>
</tr>
<tr>
<td>hash_small8</td>
<td>2.252</td>
<td>2.158</td>
<td>2.341</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>1.031</td>
<td>1.045</td>
<td>1.010</td>
</tr>
<tr>
<td>hash_values</td>
<td>2.864</td>
<td>1.305</td>
<td>1.626</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>3.323</td>
<td>7.471</td>
<td>5.769</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base+hash+strong hash patches:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.580</td>
<td>1.169</td>
<td>1.097</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>0.968</td>
<td>0.962</td>
<td>1.015</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.428</td>
<td>1.501</td>
<td>1.333</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>1.047</td>
<td>0.977</td>
<td>1.146</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.917</td>
<td>1.052</td>
<td>1.398</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>1.236</td>
<td>1.424</td>
<td>1.168</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.346</td>
<td>1.451</td>
<td>1.123</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.000</td>
<td>0.913</td>
<td>0.989</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.049</td>
<td>0.936</td>
<td>1.055</td>
</tr>
</tbody>
</table>

05/08/2021
Are the first three patches OK for the trunk? If it is so, I don’t
know how to commit them to the trunk. So could somebody commit them
or explain the procedure how to do it in this case.

Only committers may commit; but developers need not be committer

https://bugs.ruby-lang.org/projects/ruby/wiki/CommitterHowto

Anyways, I’ve taken a light look at patches 1-3 and applied them
to some of my own machines. There was some whitespace warnings from
git, so I also pushed them out in the "f-12142-hash-160429" branch
to git://bogomips.org/ruby.git (or https://bogomips.org/ruby.git)

Will wait for others to have time to review/test, too, since it’s
the weekend ->

You forgot to include source for added benchmark files into diff: bm_hash_small[2,4,8], bm_bighash, bm_hash_long
And still, why don't you measure with realworld application too?

It is relatively easy to setup Redmine http://www.redmine.org/ - the same software which powers this site (bugs.ruby-lang.org). Add project, some issues and use ab (apache benchmark) to stress load it.

---

Eric Wong wrote:

vmakarov@redhat.com wrote:

Are the first three patches OK for the trunk? If it is so, I don't know how to commit them to the trunk. So could somebody commit them or explain the procedure how to do it in this case.

Only committers may commit; but developers need not be committer

https://bugs.ruby-lang.org/projects/ruby/wiki/CommitterHowto

Anyways, I've taken a light look at patches 1-3 and applied them to some of my own machines. There was some whitespace warnings from git, so I also pushed them out in the "f-12142-hash-160429" branch to git://bogomips.org/ruby.git (or https://bogomips.org/ruby.git)

Will wait for others to have time to review/test, too, since it's the weekend :>

Thank you, Eric. I don't expect a quick response. In GCC community, a review of the patches of such size can take a few weeks. I promised you to send the patches at the end of April, in order you can work on it (by the way the previous patch I sent you was buggy).

I read the documents but I can not find anything about copyright assignment procedure. Is there any? It is not a trouble for me because I am willing to give the code to whoever is the copyright holder of MRI.

As Yura wrote I missed the new benchmarks. I am adding them here. It should be part of the base patch.

---

Yura Sokolov wrote:

You forgot to include source for added benchmark files into diff: bm_hash_small(2,4,8), bm_bighash, bm_hash_long

Thank you for pointing this out, Yura. I've added them in the previous message. Apparently I missed them when I applied the patches to a new working directory and did not do git add.

---

Well, I am too far away from the world of WEB developers. I've tried and it was not easy for me. After spending some time, I decided that if it is not easy for me, it will be not easy for some other people too. The most important part of benchmarking is its reproducibility. For example, it was interesting for me when you reported a big improvement on Redmine. I was interesting what part of the improvement belongs to new hash tables and what part belongs to jemalloc usage.

After that I tried easier setup with Sinatra using puma and wrk (a benchmark client) on i7-4790K. I used 4 threads and 4 workers. The results were varying in a big range, so I did 10 runs each for 30 sec for the both hash table implementation. The best speed I got was 11450 reqs/sec for the old table and 11744 reqs/sec for the new ones. But again I'd take the results with a grain of salt as I wrote the results were varying too much for runs.

---

I've put result for each combination: with/without jemalloc * old/new hash. (2^2=4)
There are two ways to get more reproducible result:

- first way is to use just one worker process (with several threads) and use taskset to bind it to single core
- second way is to run more workers than cores and run workload (wrk for example) from other computer.

In any way, be sure that you've disabled all logging.

BTW 117k vs 114k - 3% improvement, not bad. But it is on par with my much simpler implementation.

Perhaps, your memory usage is smaller cause of small hashes, but it should be measured.

---

**#85 - 04/30/2016 08:04 PM - ko1 (Koichi Sasada)**

Yura Sokolov wrote:

I've put result for each combination: with/without jemalloc * old/new hash. (2*2=4)

where?

---

**#86 - 04/30/2016 11:58 PM - vmakarov (Vladimir Makarov)**

Koichi Sasada wrote:

Yura Sokolov wrote:

I've put result for each combination: with/without jemalloc * old/new hash. (2*2=4)

where?

I believed it is in the comment #56 of this thread.

I'd like only to add that, IMHO, switching to jemalloc by default should be a well tested decision. I read that some people reported that for their load jemalloc used >2GB memory when glibc malloc used only 900MB. Typical Facebook and Google work load can be very different from other people load. For example, GCC has features which nobody except Google needs them (whopr which is a link time optimization mode for such huge C++ programs whose IR can not fit into memory of very beefy servers).

I know that glibc people are well aware about jemalloc/tcmalloc competition and recently started to work on investigating the allocators behavior and probably malloc modernization.

---

**#87 - 05/30/2016 03:11 AM - normalperson (Eric Wong)**

Btw, I haven't forgotten about this, just haven't gotten around to looking at it, more. But I still run it daily for random minor scripts and haven't noticed incompatibilities/bugs creep up.

Anyways, I've rebased against r55206 and fixed some trivial conflicts in include/ruby/st.h

https://80x24.org/spew/20160530025812.21803-1-e%4080x24.org/t.mbox.gz

---

**#88 - 05/30/2016 11:28 PM - vmakarov (Vladimir Makarov)**

Eric Wong wrote:

Btw, I haven't forgotten about this, just haven't gotten around to looking at it, more. But I still run it daily for random minor scripts and haven't noticed incompatibilities/bugs creep up.

Anyways, I've rebased against r55206 and fixed some trivial conflicts in include/ruby/st.h

https://80x24.org/spew/20160530025812.21803-1-e%4080x24.org/t.mbox.gz

---

Thank you for the update and testing the patch. Please, don't worry. I am also busy with other different projects. There is no hurry for me to see the code on MRI mainline.

---

**#89 - 07/13/2016 09:20 AM - funny_falcon (Yura Sokolov)**

- File hash_improvements_and_st_implementation_changes.mbox added
Good day, everyone.

I've updated my version of st.c and hashing patch. Main difference in st implementation from Vladimir Makarov's patch is usage of closed addressing instead of open addressing. Code size approximately:

- my version of st.c is near 1000 lines
- Makarov's one is near 1250 lines (both without huge comments and tables of sizes, Makarov's without Quadratic probing also).

Also there are:

- fixes for Murmur (it had issues with st_hash_uint and st_hash_end) (I've also replaced Murmur1/2 to Murmur3).
- make Murmur seeded, and use it for 'identity' hashes.
- suggestion to use SipHash13 instead SipHash24 (Rust already accepted it)
- suggestion to use 32bit SipHash's cousin for 32bit architecture

Overall performance is on par with Vladimir's patch. (note: Vladimir's patch removes seeding for fixnum/double/symbol hashing, so it could be faster for keys of such types) (I've tested against Eric's Wong mailbox applied on trunk, so no CityHash)

Uploaded mailbox patch:
https://bugs.ruby-lang.org/attachments/download/6061/hash_improvements_and_st_implementation_changes.mbox

Also could be seen on github:
https://github.com/funny-falcon/ruby/compare/trunk...funny-falcon:st_table_with_array2
https://github.com/funny-falcon/ruby/compare/trunk...funny-falcon:st_table_with_array2.patch

Performance of Redmine (software that runs bugs.ruby-lang.org):

```
ab -n 1000 -c 10 http://localhost:3000/projects/general/issues
trunk
Requests per second: 27.51 [#/sec] (mean)
Requests per second: 27.54 [#/sec] (mean)
Requests per second: 27.69 [#/sec] (mean)
makarov
Requests per second: 28.29 [#/sec] (mean)
Requests per second: 28.12 [#/sec] (mean)
Requests per second: 28.30 [#/sec] (mean)
falcon
Requests per second: 28.31 [#/sec] (mean)
Requests per second: 28.16 [#/sec] (mean)
Requests per second: 28.26 [#/sec] (mean)
```

```
ab -n 1000 -c 10 http://localhost:3000/issues/1
trunk
Requests per second: 26.37 [#/sec] (mean)
Requests per second: 26.51 [#/sec] (mean)
Requests per second: 26.48 [#/sec] (mean)
makarov
Requests per second: 27.39 [#/sec] (mean)
Requests per second: 27.41 [#/sec] (mean)
Requests per second: 27.75 [#/sec] (mean)
falcon
Requests per second: 27.54 [#/sec] (mean)
Requests per second: 27.23 [#/sec] (mean)
Requests per second: 27.47 [#/sec] (mean)
```

Tests using benchmark/driver.rb (tested miniruby):

```
Speedup ratio: compare with the result of `trunk' (greater is better)
64bit + jemalloc
name    makarov    falcon
bighash  1.320    1.447
hash_aref_dsym  0.976    1.009
hash_aref_dsym_long  1.474    1.459
hash_aref_fix  1.042    1.023
hash_aref_flo  1.671    1.595
hash_aref_miss  1.014    1.154
hash_aref_str  1.023    1.123
hash_aref_sym  0.997    1.016
hash_aref_sym_long  1.014    1.005
```
<table>
<thead>
<tr>
<th></th>
<th>64bit no jemalloc</th>
<th>32bit no jemalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>makarov</td>
<td>falcon</td>
</tr>
<tr>
<td>bighash</td>
<td>1.741 1.840</td>
<td>1.103 1.267</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>0.934 0.970</td>
<td>0.991 1.014</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.546 1.519</td>
<td>1.356 1.444</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>1.045 1.041</td>
<td>0.992 1.017</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.734 1.654</td>
<td>1.175 2.051</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>0.980 1.022</td>
<td>1.017 1.468</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.016 1.068</td>
<td>1.010 1.530</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.000 1.015</td>
<td>1.021 1.051</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.017 1.004</td>
<td>1.032 1.052</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.172 1.179</td>
<td>1.073 1.122</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.955 0.987</td>
<td>1.124 1.114</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.960 0.955</td>
<td>0.900 0.914</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.952 0.947</td>
<td>0.921 0.947</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.947 0.956</td>
<td>0.908 0.945</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.990 1.005</td>
<td>0.910 0.949</td>
</tr>
<tr>
<td>hash_keys</td>
<td>2.820 2.870</td>
<td>2.811 2.850</td>
</tr>
<tr>
<td>hash_long</td>
<td>0.995 1.206</td>
<td>2.275 1.886</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.437 1.481</td>
<td>1.037 1.014</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.397 1.426</td>
<td>1.388 1.426</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.389 1.426</td>
<td>1.385 1.412</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.059 1.015</td>
<td>1.068 1.020</td>
</tr>
<tr>
<td>hash_small2</td>
<td>1.095 1.014</td>
<td>2.275 1.886</td>
</tr>
<tr>
<td>hash_small4</td>
<td>1.080 1.020</td>
<td>2.275 1.886</td>
</tr>
<tr>
<td>hash_small8</td>
<td>2.275 1.886</td>
<td>2.275 1.886</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>1.037 1.014</td>
<td>2.811 2.850</td>
</tr>
<tr>
<td>hash_values</td>
<td>3.429 3.372</td>
<td>3.429 3.372</td>
</tr>
<tr>
<td>loop_whileloop2</td>
<td>1.032 1.052</td>
<td>1.000 1.007</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>1.741 1.840</td>
<td>1.103 1.267</td>
</tr>
</tbody>
</table>

05/08/2021
hash_small4 1.044 0.961
hash_small8 1.570 1.378
hash_to_proc 1.007 0.989
hash_values 2.215 2.252
loop_whileloop2 1.002 1.000
vm2_bighash* 3.268 2.969

#91 - 09/02/2016 06:44 PM - normalperson (Eric Wong)

ko1, nobu, others anybody have time to review and compare these hash implementations?

I've been using vmakarov's patches on some of my systems without problems, but funny_falcon's numbers look promising, too.

https://bugs.ruby-lang.org/issues/12142

Thanks. I've been busy this year outside of Ruby.

#92 - 09/02/2016 07:48 PM - vmakarov (Vladimir Makarov)

Eric Wong wrote:

ko1, nobu, others anybody have time to review and compare these hash implementations?

I've been using vmakarov's patches on some of my systems without problems, but funny_falcon's numbers look promising, too.

https://bugs.ruby-lang.org/issues/12142

Thanks. I've been busy this year outside of Ruby.

The following link contains a comparison of the two implementations I did last weekend:

https://github.com/vnmakarov/ruby/blob/trunk/README.md

I might be unreachable for a few days as I'll be traveling.

#93 - 09/05/2016 01:09 AM - spinute (Satoru Horie)

Hello, everyone.

I want to help to merge these excellent results into trunk. I'm now reviewing both codes, and trying to consider other benchmarks to provide more solid evidence toward merge and to compare three implementations.

Firstly, Clang reports errors. (It seems to be caused by -Wshorten-64-to-32)
I sent PRs to both repos.

I run benchmarks on my environment. The result are following.

- Environment
  OSX10.11.6, 1.1GHz Intel Core m3
  Apple LLVM version 7.3.0 (clang-703.0.31)
  Target: x86_64-apple-darwin15.6.0
- Command (borrowed from https://github.com/vnmakarov/ruby/blob/trunk/README.md)
  ruby ../research/ht/funny_falcon_array2/benchmark/driver.rb -p hash -r 3 -e ./miniruby -e ../research/ht/funny_falcon_array2/miniruby -e ../research/ht/vlad_ruby/miniruby | awk 'NF==3 && /hash/ {s+=$2;s2+=$3;n++;print} END{print s/n, s2/n}'
- Targets
  funny_falcon's: e38ae901c443370186b57f5b88359f15d25974a0 + fix compilation error (brench st_table_with_array2)
  Vladimir's: 314c25696cc97a9e44c493f5c6f30b1ba9f52252 + fix compilation error (branch hash_table_with_open_addressing)
  trunk: e5c6454efa01aedd44bc59a5f32d5f1b872dd5

bighash 1.408 1.145
hash_aref_dsym 0.996 0.969
hash_aref_dsym_long 1.914 1.894
hash_aref_fix 1.013 1.007
hash_aref_flo 1.628 1.725
hash_aref_miss 0.919 1.010
hash_aref_str 1.061 1.292

05/08/2021 51/73
<table>
<thead>
<tr>
<th>Function</th>
<th>Time Old (s)</th>
<th>Time New (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_aref_sym</td>
<td>1.023</td>
<td>0.983</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.111</td>
<td>1.061</td>
</tr>
<tr>
<td>hashflatten</td>
<td>1.097</td>
<td>1.071</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.965</td>
<td>0.941</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.936</td>
<td>0.940</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.918</td>
<td>0.923</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.900</td>
<td>0.912</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.991</td>
<td>0.958</td>
</tr>
<tr>
<td>hash_keys</td>
<td>1.780</td>
<td>1.744</td>
</tr>
<tr>
<td>hash_long</td>
<td>1.194</td>
<td>1.532</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.758</td>
<td>1.670</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.760</td>
<td>1.634</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.818</td>
<td>1.692</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.823</td>
<td>1.670</td>
</tr>
<tr>
<td>hash_small12</td>
<td>1.182</td>
<td>1.049</td>
</tr>
<tr>
<td>hash_small14</td>
<td>1.124</td>
<td>1.010</td>
</tr>
<tr>
<td>hash_small18</td>
<td>1.832</td>
<td>1.999</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>0.984</td>
<td>1.108</td>
</tr>
<tr>
<td>hash_values</td>
<td>1.780</td>
<td>1.825</td>
</tr>
<tr>
<td>vm2_bighash</td>
<td>4.304</td>
<td>5.667</td>
</tr>
<tr>
<td></td>
<td>1.41552</td>
<td>1.4567</td>
</tr>
</tbody>
</table>

I may ask some questions on implementations and on a requirement of other benchmarks after my reviewing.

### #94 - 09/05/2016 12:08 PM - funny_falcon (Yura Sokolov)

- File `hash_improvements_and_st_implementation_changes.mbox` added

Vladimir's branch sometimes faster cause it uses simplified (but fast on 64bit) unseeded hash for FIXNUM and FLOAT. imho, it is security issue. I've tried to make change murmur into more correct and "secure" thing instead.

But sometimes, it is really algorithmic differences (ie. sometimes open-addressing better, sometimes chaining better).

I hope, using SipHash13 will be accepted regardless of st.c algorithmic path. It is really safe change.

And I think, other my changes to hashsum computations are reasonable also (independently of hash-table algorithm). But they should be reviewed.

I've rebased patch and fix compilation with clang.

### #95 - 09/05/2016 12:08 PM - funny_falcon (Yura Sokolov)

- File deleted (`hash_improvements_and_st_implementation_changes.mbox`)

### #96 - 09/06/2016 09:12 PM - vmakarov (Vladimir Makarov)

Satoru Horie wrote:

I run benchmarks on my environment. The result are following.

Thank you for benchmarking. I am just wondering, did you use non-zero macro `ENABLE_HUGEHASH` when you compiled Yura's code?

I am asking this because Yura's patch by default permits tables with maximum $2^{32}$ elements while my patch permits potentially tables with $2^{64}$ elements.

To compare apples to apples, I believe you should use this non-zero macro. It will definitely change the result as it will make Yura's table elements size by 33% bigger in comparison with the elements size of Yura's smaller default tables and my implementation.

- Environment OSX10.11.6, 1.1GHz Intel Core m3 Apple LLVM version 7.3.0 (clang-703.0.31) Target: x86_64-apple-darwin15.6.0

I never benchmarked MRI compiled by LLVM vs one compiled by GCC. It would be interesting for me to see such comparison.

Knowing GCC and LLVM internals, I think that GCC should generate a faster code for interpreters than LLVM as GCC uses a modification of Chaitin-Briggs register allocator.
while LLVM uses extended linear scan RA. Chaitin-Briggs RA better fits to programs with complicated CFG (like a big switch or code with computed gotos used for VM insn execution). Therefore GCC always beats LLVM on perl benchmarks of SPEC2000/SPEC2006 performance test suite.

Of course, GCC should be fresh, not outdated GCC-4.2 provided by Apple which uses worse Chow's priority-based RA.

Yura Sokolov wrote:

   Vladimir's branch sometimes faster cause it uses simplified (but fast on 64bit) unseeded hash for FIXNUM and FLOAT. imho, it is security issue.

Hashing numbers on the trunk is based on shifts and logical operations. My hash for numbers is faster not only on 64-bit but on 32-bit targets too, even with embedded shifter as in ARM when an operand of an arithmetic insn can be preliminary shifted during the insn execution.

I think, potentially, there is a security issue if an application getting strings, transforms them into numbers and uses them as keys. But I don't know such Ruby applications.

My implementation itself does not introduce the security issue as the original code on trunk has the same problem. Moreover producing conflicting integer keys for the trunk code is very easy while for my hash of integers it will take several months on modern supercomputers to produce such keys for integers on 64-bit machines.

Still I can make a patch quickly which completely eliminates this issue without any slowdown in my hashing for numbers if people decide that it is a real security issue. Instead of a constant seed used for now, I can produce a seed once per MRI execution from crypto-level PRNG. Moreover, I can speed up hashing for numbers even more for the last 4 generations of Intel CPUs.

Please, do it now. So no one will ever argue.

while for my hash of integers it will take several months on modern supercomputers

May you give a link to paper studying problem of cracking algorithm you use? I really like to read it for self education.

BTW, I doubt its faster (than st_hash_uint) when compiled as 32bit x86. But I agree, it doesn't really matter which is faster, cause almost no one runs Ruby on 32bit this days.

Still, my fix to st_hash (especially st_hash_uint and st_hash_end) are valuable, and they can coexist with your patch. Changing SipHash to SipHash13 is also valid and also independent of hash table algo.

I belive, Ruby core team will take best from both patches.

Friends?

Python landed same dict algo for 3.6
https://mail.python.org/pipermail/python-dev/2016-September/146327.html (same to Vladimir's approach)

And PyPy did it more than year ago
https://morepypy.blogspot.ru/2015/01/faster-more-memory-efficient-and-more.html?m=1
Hope, decision will be done, and new hash implementation will be accepted for Ruby 2.4.

#100 - 09/11/2016 08:50 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

> Still I can make a patch quickly which completely eliminates this issue without any slowdown in my hashing for numbers

Please, do it now. So no one will ever argue.

After some thoughts, I believe the security issue (collision exploitation) we are discussing is not a problem for my implementation.

First of all, we have a translation of N-bit value into N-bit hash where N=64 is most interesting. For a quality hash function, it is mostly 1-to-1 function. I can not measure maximum number of collisions for N=64 but for 32, after half hour on my desktop machine, I got maximum collision number equal to 15 for 32-bit variant of my function using 32x32->64 bit multiplication. So hypothetically, even an attacker spends huge CPU time on 64-bit machine to get keys with the same hash, I guess, he will produce a small number of collisions for the full hash.

There is still an issue for a table with chains as it uses a small part of hash (M bits) to choose the right bin, it makes an actual hash function to map \(2^N\) values into \(2^M\) values where \(N > M\). It is easy to generate a lot of keys with the same least significant M-bits for the original function (\(k_{>3} \oplus k_{>11}\)).

In the table with open addressing the M-bits of N-bits hash is used only initially but after each collision, the other bits of hash are used until all hash bits are consumed.

Changing SipHash to SipHash13 is also valid and also independent of hash table algo.

I don't think it is that important. I tried your tables with siphash13 and siphash24 and for siphash13 the average improvement of the hash table benchmarks increased only by 1% (out of 37%).

#101 - 09/11/2016 09:05 PM - funny_falcon (Yura Sokolov)

So hypothetically, even an attacker spends huge CPU time

Why not GPU time? Why not FPGA? It is really not that expensive. Edit: and there is no real need in exact 64bit match to forge python style open addressing.

Changing SipHash to SipHash13 is also valid and also independent of hash table algo.

I don't think it is that important.

And I think, it is.

Still, there is no time to force discussion between us: between me and Vladimir. We both already have said all we have to say. It is time for Ruby Core team to look at, to choose and to decide.

If new hash table will be part of Ruby2.4, then it will be great for all. Otherwise... it will be not such great.

#102 - 09/14/2016 11:31 AM - naruse (Yui NARUSE)

Current patches are below branches? (though they have -Wshorten-64-to-32 issue)

- https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing

05/08/2021
I want people to check whether increased memory consumption is acceptable.

#103 - 09/14/2016 06:07 PM - funny_falcon (Yura Sokolov)
Correct link: https://github.com/funny-falcon/ruby/tree/st_table_with_array2

Also it is attached to this issue as mbox:
https://bugs.ruby-lang.org/attachments/download/6131/hash_improvements_and_st_implementation_changes.mbox

I thought, I’ve fixed -Wshorten-64-to-32 issue nine days ago.

Does my version leads to increased memory consumption? I change array size a bit more often than 2x, so there should not be much unused allocated space. (It could be done with Vladimir's patch also after some refactoring)

#104 - 09/14/2016 09:26 PM - vmakarov (Vladimir Makarov)

Yui NARUSE wrote:

Current patches are below branches? (though they have -Wshorten-64-to-32 issue)
- https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing

Yes.

I might be unresponsive until next Monday (I am traveling). Sorry, if it happens.

#105 - 09/26/2016 05:30 PM - funny_falcon (Yura Sokolov)

- File hash_improvements_and_st_array_with_open_addressing.mbox added

#106 - 09/26/2016 06:27 PM - funny_falcon (Yura Sokolov)

Good day, everyone.

I'm adding alternative patch version for st_table.
It is my compromise with Vladimir and his proposals.
I really count and mention Vladimir Makarov as co-author of this version.

- It uses open-addressing with mixing all hash bits at start (but a bit simpler then LCG). Although I still think chaining is more suitable, code for open-addressing looks clearer.
- While I still have compilation option for "huge" hash, it affects only sizeof(st_table), layout of entries array is similar both for "maximum 32bit" and "almost unlimited" versions.
- I almost copy "use weak hash and fallback to strong on attack" suggestion (with a bit simpler check).

Pros of my version:
- less code (and I think, it is more readable, but it is just my opinion). My version of st_table algorithm has ~561 meaningful lines, Vladimir's version is ~803 lines.
- allocation size changes more frequently then x2, and tries to satisfy allocator pattern (jemalloc)
- smaller st_table
  - due to trick with one pointer for bins and entries,
  - no additional pointer for "current hash function"
  - version with 32bit index
- lazier entries compation and rehashing on table growth
- "attack" detection is not by full hash equality, but with "collision chain is statistically too long" (simpler to implement).
- improvements to st_hash* functions and siphash (it could be applied to Vladimir's patch also).

Cons of my version:
- It doesn't mark deleted entries in bins array. It allows simplify code significantly, and usually doesn't matter. But may lead to a bit more cache misses on insertion into hash with deleted entries.
- Single pointer to bins and entries needs single allocation. It leads to more coarse allocations, and could be disadvantage. And it is a trick, that could be considered disgusting.
- I do not make separate "reserving" search for st_insert, cause st_insert is not used in Hash. (st_update of Vladimir's version also do not use reservation). It simplifies code, but could be slower in some other places. It is not hard to implement it, if considered necessary.
- "statistically too long collision chain" may happen occasionally, though, it is unlikely.
- no much comments.

https://github.com/funny-falcon/ruby/tree/st_table_with_array3
https://bugs.ruby-lang.org/attachments/download/6159/hash_improvements_and_st_array_with_open_addressing.mbox

I didn't run benchmark.
I've measured by making documentation for ruby, and my version were on par with Vladimir's.
(and yes, it was really hard to make it close to).
(hope benchmark will not be shameful :-} )

#107 - 09/27/2016 07:04 AM - funny_falcon (Yura Sokolov)
- File hash_improvements_and_st_array_with_open_addressing.mbox added

#108 - 09/27/2016 07:04 AM - funny_falcon (Yura Sokolov)
- File deleted (hash_improvements_and_st_array_with_open_addressing.mbox)

#109 - 09/27/2016 07:06 AM - funny_falcon (Yura Sokolov)
tiny fix for string hash function.
branch is updated with force.
mbox re-uploaded: https://bugs.ruby-lang.org/attachments/download/6160/hash_improvements_and_st_array_with_open_addressing.mbox

#110 - 09/29/2016 03:50 PM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

Good day, everyone.
I'm adding alternative patch version for st_table.
It is my compromise with Vladimir and his proposals.

Yura, I am glad that finally you accepted open address tables. Now
there is no major differences between my original table proposal and
your current implementation. Although there are differences in details.

I don't like idea adding more and more new things to the patch without
separate evaluation of them. Therefore I stopped to change my patch
five months ago and only did a rebase. Effect of my 4 major parts of
the patch on performance were evaluated and posted by me to confirm
their inclusion or not.

IMHO, Ruby team should accept some base patch (in case of my patch it
is what was done 5 months ago and currently on
https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing)
and after that additional patches from you or/and me could be considered
and evaluated.

The base patch could be what they already evaluated
or agree to evaluate. Otherwise, you and me will create new "better" versions
of the patch creating stress to them. Adding new features separately is also
good for keeping this thread readable (I suspect we made a record for the longest
discussion on this Ruby discussion board).

Your last new features are interesting but some of them is not obvious
to me, for example, using tables with less 2^32 by default or
faster hash table grow as you wrote for jemalloc allocation pattern
(As I know MRI Ruby does not use jemalloc yet and I am not sure it
should be used because it uses more memory for some loads and also
because Glibc community is serious to improve their malloc --

#111 - 09/29/2016 07:29 PM - funny_falcon (Yura Sokolov)
I still don't like open-addressing here. But code looks clearer.
And it is better to have common than different when Christmas is so close.

Ruby team will accept (or not accept) whatever they like to accept (or to not accept).
I don't expect they will take any specific decision.

With regards,
Yura.

#112 - 10/11/2016 11:59 AM - shyouhei (Shyouhei Urabe)
We (mainly ko1) had a deeper look at this issue in developer meeting today. Ko1 showed his microbenchmark result over the three (proposed two
and the current) implementations. I don't have the result at hand right now so let me refrain from pointing the winner. I believe he will reveal his
benchmark script (and hopefully the result).

What I can say now is that he found the most use of hash tables in a Rails application was lookups of small (entries <= 64) hashes. So to speed up
real-world applications, that kind of usage shall be treated carefully.
Anyways, we (at least myself) would like to accept either one of proposed hash implementations and in order to do so, ko1 is measuring which one is better. Thank you for all your efforts. Please let us give a little more time.

#113 - 10/31/2016 03:40 AM - shyouhei (Shyouhei Urabe)
- Related to Feature #5903: Optimize st_table (take 2) added

#114 - 11/02/2016 09:28 AM - ko1 (Koichi Sasada)
Sorry for my lazy-ness, I evaluated your two implementations (ffalcon has two versions, HUGEHASH or not).

Now, I depicted all of my evaluations in charts, so I introduce this report: https://gist.github.com/ko1/dda8243134bc40f7fc05a293abc544b2#file-report-md
I'll add explanation of this evaluation soon.

#115 - 11/02/2016 04:45 PM - vmakarov (Vladimir Makarov)
Koichi Sasada wrote:

Sorry for my lazy-ness, I evaluated your two implementations (ffalcon has two versions, HUGEHASH or not).

Now, I depicted all of my evaluations in charts, so I introduce this report: https://gist.github.com/ko1/dda8243134bc40f7fc05a293abc544b2#file-report-md
I'll add explanation of this evaluation soon.

Koichi, thank you for your detail investigation of the two tables. Looking at this report, I got an impression that you spent more time on this report than we spent time on our implementations.

For the record, here is my comparison the same tables on different tests (MRI hash table benchmarks) as a speed up ratio vs trunk:

<table>
<thead>
<tr>
<th></th>
<th>Yura</th>
<th>Yura64</th>
<th>My</th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.657</td>
<td>1.751</td>
<td>1.612</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>0.929</td>
<td>0.927</td>
<td>0.967</td>
</tr>
<tr>
<td>hash_aref_dsym_long</td>
<td>1.314</td>
<td>1.444</td>
<td>1.427</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>0.981</td>
<td>0.974</td>
<td>1.051</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.662</td>
<td>1.656</td>
<td>1.908</td>
</tr>
<tr>
<td>hash_aref_mess</td>
<td>1.213</td>
<td>1.195</td>
<td>1.209</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.217</td>
<td>1.250</td>
<td>1.338</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>0.924</td>
<td>0.932</td>
<td>0.992</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.001</td>
<td>1.003</td>
<td>1.042</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.082</td>
<td>1.091</td>
<td>1.184</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>0.912</td>
<td>0.919</td>
<td>0.970</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.924</td>
<td>0.918</td>
<td>0.971</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.878</td>
<td>0.876</td>
<td>0.963</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.902</td>
<td>0.896</td>
<td>0.960</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.952</td>
<td>0.950</td>
<td>0.978</td>
</tr>
<tr>
<td>hash_keys</td>
<td>2.730</td>
<td>2.806</td>
<td>2.813</td>
</tr>
<tr>
<td>hash_long</td>
<td>1.439</td>
<td>1.463</td>
<td>1.564</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.297</td>
<td>1.283</td>
<td>1.403</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.229</td>
<td>1.325</td>
<td>1.387</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.224</td>
<td>1.201</td>
<td>1.378</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.220</td>
<td>1.201</td>
<td>1.376</td>
</tr>
<tr>
<td>hash_small12</td>
<td>1.081</td>
<td>0.985</td>
<td>1.040</td>
</tr>
<tr>
<td>hash_small14</td>
<td>1.055</td>
<td>0.980</td>
<td>1.050</td>
</tr>
<tr>
<td>hash_small18</td>
<td>2.006</td>
<td>2.009</td>
<td>2.358</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>0.981</td>
<td>0.973</td>
<td>1.027</td>
</tr>
<tr>
<td>hash_values</td>
<td>2.817</td>
<td>2.790</td>
<td>2.823</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>2.423</td>
<td>2.180</td>
<td>3.264</td>
</tr>
<tr>
<td></td>
<td>1.33519</td>
<td>1.33252</td>
<td>1.44648</td>
</tr>
</tbody>
</table>

I used the following script on 4.2GHz i7-4790K for this:

```
ruby ./ruby/benchmark/driver.rb -p hash -r 3 -e trunk::trunk/miniruby -e yura::yura/miniruby -e yura::yura64/miniruby -e current::./miniruby 2>/dev/null
ull|awk 'NF==4 && /hash/ {s1+=$2;s2+=$3;s3+=$4;p++;print} END{print s1/n, s2/n, s3/n}''
```

You wrote "I agree to determine the implementation with coin toss" in your report. I think it is better not do this.

My implementation is the original one. Yura just repeated my major ideas: putting elements in array, open hash addressing (although he fiercely arguing with me a lot that the buckets are better), using
weaker hash function when there is no suspension of a collision attack.
I did not use any his ideas.

In fact I stopped to improve my tables in April, Yura produced his
latest variant in September (btw it would be interesting to see also
comparison of his other two variants with buckets he proposed in July.
I put some such comparison on

I believed the competition was unhealthy and it seems I was right as
Yura spent his time on his implementation and you had to spend your time on your thorough
investigation and still you need "a coin toss" to decide.

I'd like to propose a plan which is to use my code as the base and
Yura can add his own original code later:

- fixing Murmur hash in MRI
- different hash tables growth rate (it needs performance evaluation)
- smaller hash table header (usually it is accessed randomly and my
header is just one cache line, further size decreasing hardly improves
performance, imho)
- statistical collision attack recognition (i doubt it will improves
performance as the surrounded code is memory bound)

Again this competition was unhealthy at least for me. I could have
spent more time on more serious MRI performance improvement.

I'll respect any your decision, it will be enough for me that all my
major ideas about MRI hash tables is used.

#116 - 11/02/2016 10:53 PM - funny_falcon (Yura Sokolov)
- File hash_improvements_additional.mbox added

#117 - 11/02/2016 11:36 PM - funny_falcon (Yura Sokolov)
- File deleted (hash_improvements_additional.mbox)

#118 - 11/02/2016 11:41 PM - funny_falcon (Yura Sokolov)
- File hash_improvements_additional.mbox added

#119 - 11/03/2016 12:35 AM - funny_falcon (Yura Sokolov)
Koichi, thank you for investigation.
Your graphs give a lot of useful information.

Expected behaviour:

- memory usage is lesser with my patch (cause it has more reallocation steps, and smaller header)
- performance of huge table is worse a bit than with Vladimirs patch (cause my patch allows larger fill ratio of bins)
- performance of "many" tables is usually better (cause of smaller header and single pointer to bins+entries, that leads to lesser TLB misses)

Unexpected behaviour:

- unstable performance of single small table.
- my attempts to make murmur strong makes it remarkably slower.

Using this information I added couple of commits:

- first commit switches table to version with "bins" earlier. It improves performance of tables of size 5-8, and reduces size of tables 9-10. Although,
tables of size 8 are larger a bit, and tables of size 10 could be a bit slower.
- second commit improves performance of "murmur". "Murmur" still remains seeded with random seed, and it is not as trivial for flood as original
murmur, but it becomes weaker than previous variant. Additionally, read of string tail improved for cpu with unaligned word access.

mbox patch:
https://bugs.ruby-lang.org/attachments/download/6210/hash_improvements_additional.mbox

(branch on github is updated and rebased on trunk:
https://github.com/funny-falcon/ruby/commits/st_table_with_array3
It contains additional commit with adding rb_str_hash_weak
for symhash and rb_fstring_hash_type.
It improves "#{dynamic}".to_sym a bit.
But this commit doesn't affect your benchmark or bm_hash",
so I didn't include it to mbox)
Hello, the following patch contains a new implementation of hash tables (major files st.c and include/ruby/st.h).

Modern processors have several levels of cache. Usually, the CPU reads one or a few lines of the cache from memory (or another level of cache). So CPU is much faster at reading data stored close to each other. The current implementation of Ruby hash tables does not fit well to modern processor cache organization, which requires better data locality for faster program speed.

The new hash table implementation achieves a better data locality mainly by

- switching to open addressing hash tables for accessing by keys. Removing hash collision lists lets us avoid "pointer chasing", a common problem that produces bad data locality. I see a tendency to move from chaining hash tables to open addressing hash tables due to their better fit to modern CPU memory organizations. CPython recently made such switch (https://hg.python.org/cpython/file/ff1938d12240/Objects/dictobject.c). PHP did this a bit earlier https://nikic.github.io/2014/12/22/PHPs-new-hashtable-implementation.html. GCC has widely-used such hash tables (https://gcc.gnu.org/svn/gcc/trunk/libiberty/hashtab.c) internally for more than 15 years.

- removing doubly linked lists and putting the elements into an array for accessing to elements by their inclusion order. That also removes pointer chasing on the doubly linked lists used for traversing elements by their inclusion order.

A more detailed description of the proposed implementation can be found in the top comment of the file st.c.

The new implementation was benchmarked on 21 MRI hash table benchmarks for two most widely used targets x86-64 (Intel 4.2GHz i7-4790K) and ARM (Exynos 5410 - 1.6GHz Cortex-A15):

```
make benchmark-each ITEM=bm_hash OPTS='-r 3 -v' COMPARE_RUBY='<trunk ruby>'
```

Here are the results for x86-64:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_arref_dsym</td>
<td>1.094</td>
</tr>
<tr>
<td>hash_arref_dsym_long</td>
<td>1.383</td>
</tr>
<tr>
<td>hash_arref_fix</td>
<td>1.048</td>
</tr>
<tr>
<td>hash_arref_flo</td>
<td>1.860</td>
</tr>
<tr>
<td>hash_arref_miss</td>
<td>1.107</td>
</tr>
<tr>
<td>hash_arref_str</td>
<td>1.107</td>
</tr>
<tr>
<td>hash_arref_sym</td>
<td>1.191</td>
</tr>
<tr>
<td>hash_arref_sym_long</td>
<td>1.113</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.258</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>1.627</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>1.045</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>1.143</td>
</tr>
</tbody>
</table>
The average performance improvement is more than 50%. ARM results are analogous -- no any benchmark performance degradation and about the same average improvement.

The patch can be seen as

https://github.com/vnmakarov/ruby/compare/trunk...hash_tables_with_open_addressing.patch

or in a less convenient way as pull request changes

https://github.com/ruby/ruby/pull/1264/files

This is my first patch for MRI and may be my proposal and implementation have pitfalls. But I am keen to learn and work on inclusion of this code into MRI.

---Files--------------------------------
0001-st.c-change-st_table-implementation.patch (59.4 KB)
st-march31.patch (114 KB)
base.patch (93.8 KB)
hash.patch (4.48 KB)
strong_hash.patch (8.08 KB)
city.patch (19.4 KB)
new-hash-table-benchmarks.patch (1.34 KB)
hash_improvements_and_st_array_with_open_addressing.mbox (108 KB)
hash_improvements_additional.mbox (5.09 KB)

#121 - 11/03/2016 01:23 AM - ko1 (Koichi Sasada)
Yura:

Could you check that there are no memory usage (st_memsize()) between HUGE=0 and HUGE=1. Is it reasonable?
Here is a raw data https://gist.github.com/ko1/dda8243134bc40f7fc05e293abc5f4b2#file-size-csv.

#122 - 11/03/2016 03:46 AM - vmakarov (Vladimir Makarov)
As small table size seems so important now, I did some small changes to my branch to decrease it.

https://github.com/vnmakarov/ruby/commit/a99e4e0e6929af225c2b7475fa6b92e989bc3b07

Before the change, my code maintained 4:1 ratio of bins vs entries for up to $2^{31}$ elements tables for 64-bit targets (up to $2^{15}$ for 32-bit targets). Now the ratio is 2.

The change decreases table size:

- by 5% for tables up to 128 elements
- by 12.5% for tables up to 16K elements
- by 20% for tables up to $2^{21}$ elements

It is not for free, the price is about 2% decrease of an average performance of MRI hash table benchmarks. Still this performance should be about 9% better than one of Yura's patch.

I am going to work further to reduce size of the small tables.

#123 - 11/03/2016 08:45 AM - funny_falcon (Yura Sokolov)

Koichi,

Difference between HUGE=0 and HUGE=1 is small and constant cause they differs only in header ( struct st_table ).
HUGE=1 has 48 byte header and HUGE=0 has 32 byte header.

Difference were larger when "chaining" were used (instead of open addressing), cause then st_table_entry had different structure for HUGE=0 and HUGE=1:

```c
// chaining, HUGE=0
struct st_table_entry {
    uint32_t hash, next;
    uint64_t key, record;
};

// chaining, HUGE=1
struct st_table_entry {
    uint64_t hash, next;
    uint64_t key, record;
};

// openaddressing, independently of HUGE
struct st_table_entry {
    uint64_t hash;
    uint64_t key, record;
};
```

I believe, chaining is better.
But difference in size, and quirks with maintaining "optimized" bins on deletion convinced me to switch to open-addressing.

If you wish, I may attempt to make "chaining" again on top of current code.
There is not big difference between code bases.
I think, using same trick on deletion as in open addressing (don't bothering with maintaining exact bin), code could be as clear.
But then table will use 30% more memory on 64bit with HUGE=1 and always on 32bit.

Vladimir:
I'm glad you returned to competition. Together we will make st_table great!

    Before the change, my code maintained 4:1 ratio of bins vs entries for upto 2^31 elements

Now I understand, why performance of your tables were much stabler.
It is logical result of 25% max bins fill rate.

#124 - 11/03/2016 11:26 AM - vmakarov (Vladimir Makarov)
Yura Sokolov wrote:

    Vladimir:

        I'm glad you returned to competition. Together we will make st_table great!

To be honest I am not glad. I am working on RTL VM insns and JIT and it is a big distraction for me.
Instead of accepting one (original) variant and then improving it (as I proposed yesterday), here we have more variants and more variants coming as there is no deadline of this stupid competition without exact rules.
Koichi did a big job comparing already existing tables. Now this job should be done again and again especially when you are returning back saying that "I believe, chaining is better." (which means at least 2 more variants from you).

    Before the change, my code maintained 4:1 ratio of bins vs entries for upto 2^31 elements

Now I understand, why performance of your tables were much stabler.
It is logical result of 25% max bins fill rate.

It gives only 2% improvement on MRI hash tables benchmarks. There are other reasons why still my tables 9% faster than yours on these benchmarks even after my last change.

#125 - 11/03/2016 01:39 PM - funny_falcon (Yura Sokolov)
- File return-chaining-into-implementation.patch added

#126 - 11/03/2016 01:42 PM - funny_falcon (Yura Sokolov)
- File deleted (return-chaining-into-implementation.patch)
#127 - 11/03/2016 01:45 PM - funny_falcon (Yura Sokolov)
- File return-chaining-into-implementation.patch added

#128 - 11/03/2016 01:51 PM - funny_falcon (Yura Sokolov)

Koichi,

Here is patch that returns chaining in my implementation.
It is smaller than I expected :-)  
https://bugs.ruby-lang.org/attachments/download/6212/return-chaining-into-implementation.patch

(it is applied after last additional two commit in hash_improvements_additional.mbox)

It has better performance than both my and Vladimir's open-addressing variants.  
It will consume same amount of memory on 64bit with HUGE=0 as my open-addressing variant.  
It will consume 30% more memory on 32bit or HUGE=1.

Github branch:
https://github.com/funny-falcon/ruby/commits/st_table_with_array4

#129 - 11/03/2016 02:16 PM - funny_falcon (Yura Sokolov)

Well, chaining is not faster with HUGE=1. It is only faster with HUGE=0.

#130 - 11/03/2016 03:01 PM - ko1 (Koichi Sasada)

My assumption is HUGE enables to extend 64bits entries and HUGE entries  
are increase constantly. And table size should be proportion of entry  
count. However st_memsize() returns only constant difference compare  
with HUGE=0.

So I asked about it. And I can't solve the question, yet.

--  
// SASADA Koichi at atdot dot net

#131 - 11/03/2016 06:24 PM - funny_falcon (Yura Sokolov)

Koichi,

HUGE makes difference for st_table_entry only for version with chaining.  
Version with open addressing has same st_table_entry regardless of HUGE.  
It differs only in sizeof st_table.

You had tested this version with open addressing:
https://bugs.ruby-lang.org/attachments/download/6160/hash_improvements_and_st_array_with_open_addressing.mbox

Yesterday I added couple of commit to it:  
https://bugs.ruby-lang.org/attachments/download/6210/hash_improvements_additional.mbox

It improves performance of smallest tables and fixes performance of string hashing.

Today I present optional commit that returns chaining:  
https://bugs.ruby-lang.org/attachments/download/6212/return-chaining-into-implementation.patch

With this commit size of table will react on HUGE in non-constant way.  
With HUGE=1 it will be 30% larger.  
And on 32bit platform it is larger then in open-addressing version.  
But with HUGE=0 it is fastest version on x86_64.

With regards,  
Sokolov Yura

#132 - 11/04/2016 02:37 AM - vmakarov (Vladimir Makarov)

To decrease sizes of very small tables I tuned some parameters. Now minimal table size is 4 (instead of 8) and bins starts with 16 (instead of 8):  
https://github.com/vmakarov/ruby/commit/f4d2bf3ab78e59118e5dd89bacc5c3d8c4371ed35

I did not find any meaningful changes in average performance on MRI hash table benchmarks.

I also merged last changes on the trunk to my branch.

I'll post table size graphs tomorrow (my benchmarking machine is currently busy for gcc benchmarking). It takes about 20 min for building tables of sizes from 1 to 60K.
I don't know how we will benchmark the speed as on my estimation Koichi's script will take about 2 hours on my fastest machine for one variant. As I understand now we need to test 7 variants (trunk, my April and my current variants, and 4 Yura's). I probably will not do it as I can not reserve my machine exclusively for this work for such long time. Sorry.

#133 - 11/04/2016 07:06 AM - funny_falcon (Yura Sokolov)
Vladimir is right.

Despite all my believes, I suggest again to concentrate on open-addressing variants.

I'll prepare new mbox with last changes accurately squashed into right commits and further simplified code.

I'll remove my previous variants from attachments of this issue.

#134 - 11/04/2016 08:58 PM - funny_falcon (Yura Sokolov)
- File deleted (0001-st.c-change-st_table-implementation.patch)

#135 - 11/04/2016 08:58 PM - funny_falcon (Yura Sokolov)
- File deleted (hash_improvements_and_st_implementation_changes.mbox)

#136 - 11/04/2016 08:59 PM - funny_falcon (Yura Sokolov)
- File deleted (hash_improvements_and_st_array_with_open_addressing.mbox)

#137 - 11/04/2016 08:59 PM - funny_falcon (Yura Sokolov)
- File deleted (hash_improvements_additional.mbox)

#138 - 11/04/2016 08:59 PM - funny_falcon (Yura Sokolov)
- File deleted (return-chaining-into-implementation.patch)

#139 - 11/04/2016 09:01 PM - funny_falcon (Yura Sokolov)
- File st_table_array.mbox added

#140 - 11/04/2016 09:16 PM - funny_falcon (Yura Sokolov)
- File deleted (st_table_array.mbox)

#141 - 11/04/2016 09:23 PM - vmakarov (Vladimir Makarov)
- File size60000.png added
- File size256.png added
- File size16.png added

Koichi's investigation reported that small hash tables are majority during work of Ruby on Rails:

https://gist.github.com/ko1/dda8243134bc40f7fa05e293abc5f4b2#file-report-md

Memory footprint of cloud applications like Ruby on Rails is important topic.

Since Koichi's report some changes have been done in my
(https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing)
and Yura's
(https://github.com/funny-falcon/ruby/tree/st_table_with_array3)
implementations.

Here are memory consumption graphs. The data produced by Koichi's script

https://gist.githubusercontent.com/ko1/dda8243134bc40f7fa05e293abc5f4b2/raw/dfccc2cc5e6f4749c8a9390eae3104da84eb32f/stbench.rb

on 4.2 GHz i7-4790K under FC24.

!size16.png!
!size256.png!
!size60000.png!

On these graphs
'vlad current' means my current variant (https://github.com/vnmakarov/ruby.git, branch hash_tables_with_open_addressing, f4d2b3ab78e59118e5dd89bcb5c3d8c4371ed35).

'vlad april' means my April variant without the last two commits to st.c.

trunk means the current trunk (03c9bc2)

yura means latest Yura's variant (https://github.com/funny-falcon/ruby.git, branch st_table_with_array3, 39d9b2c3658c285fbede826d1c7aeba6ba9) of tables with open addressing. I used a 64-bit variant, there is no practically difference in performance and memory consumption with 32-bit variant.

Here are the tables speed improvements relative to the trunk on MRI hash benchmarks:

<table>
<thead>
<tr>
<th></th>
<th>Yura</th>
<th>My April</th>
<th>My current</th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.716</td>
<td>1.509</td>
<td>1.443</td>
</tr>
<tr>
<td>hash_arenf_dsym</td>
<td>1.019</td>
<td>1.018</td>
<td>0.994</td>
</tr>
<tr>
<td>hash_arenf_dsym_11.394</td>
<td>1.414</td>
<td>1.377</td>
<td></td>
</tr>
<tr>
<td>hash_arenf_fix</td>
<td>0.964</td>
<td>1.041</td>
<td>1.015</td>
</tr>
<tr>
<td>hash_arenf_flo</td>
<td>1.713</td>
<td>1.832</td>
<td>1.828</td>
</tr>
<tr>
<td>hash_arenf_miss</td>
<td>1.117</td>
<td>1.191</td>
<td>1.131</td>
</tr>
<tr>
<td>hash_arenf_sym</td>
<td>1.177</td>
<td>1.225</td>
<td>1.222</td>
</tr>
<tr>
<td>hash_arenf_sym_1</td>
<td>0.959</td>
<td>1.015</td>
<td>1.012</td>
</tr>
<tr>
<td>hash_arenf_sym_2</td>
<td>1.047</td>
<td>1.066</td>
<td>1.042</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.165</td>
<td>1.095</td>
<td>1.087</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>1.005</td>
<td>0.949</td>
<td>0.958</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.932</td>
<td>0.979</td>
<td>0.954</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>0.919</td>
<td>0.942</td>
<td>0.919</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.985</td>
<td>1.000</td>
<td>0.967</td>
</tr>
<tr>
<td>hash_keys</td>
<td>3.024</td>
<td>3.039</td>
<td>3.073</td>
</tr>
<tr>
<td>hash_long</td>
<td>0.751</td>
<td>1.483</td>
<td>1.486</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.363</td>
<td>1.371</td>
<td>1.346</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.388</td>
<td>1.392</td>
<td>1.359</td>
</tr>
<tr>
<td>hash_shift_u24</td>
<td>1.344</td>
<td>1.345</td>
<td>1.310</td>
</tr>
<tr>
<td>hash_shift_u32</td>
<td>1.330</td>
<td>1.326</td>
<td>1.290</td>
</tr>
<tr>
<td>hash_small12</td>
<td>0.918</td>
<td>1.016</td>
<td>1.043</td>
</tr>
<tr>
<td>hash_small14</td>
<td>0.928</td>
<td>1.074</td>
<td>1.048</td>
</tr>
<tr>
<td>hash_small18</td>
<td>1.581</td>
<td>2.268</td>
<td>1.894</td>
</tr>
<tr>
<td>hash_to_proc</td>
<td>1.024</td>
<td>1.067</td>
<td>1.051</td>
</tr>
<tr>
<td>hash_values</td>
<td>2.801</td>
<td>2.833</td>
<td>2.822</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>2.669</td>
<td>3.261</td>
<td>3.001</td>
</tr>
<tr>
<td>Average</td>
<td>1.33841</td>
<td>1.43278</td>
<td>1.39226</td>
</tr>
</tbody>
</table>

The data are obtained by running the following script on the same machine used exclusively for benchmarking:

```
ruby ..ruby/benchmark/driver.rb -p hash -r 3 -e trunk::trunk/miniruby -e yura::yura/miniruby -e yura::yura64/ miniruby -e current::./miniruby 2>/dev/n\ ul1 awk 'NF==4 && /hash/ {s1+=$2;s2+=$3;s3+=$4;n++;print} END{print s1/n, s2/n, s3/n}''
```

Conclusions:

- My current variant requires less memory than April one. The current variant is close to average memory consumption of Yura's one (see the integral, a square of areas under the curves).
- Yura's curve is more smooth because of more frequent and smaller rebuilding but probably it results in the speed decrease.
- There is trade-off in my implementation (and probably Yura's ones) in table speed and its size.
- More size reduction in my tables by increasing table fill rate (e.g. from 0.5 to 3/4) is not worth it as one bin size is 1/24 - 1/12 of one entry size for tables with less $2^{15}$ elements.
- Unfortunately, it is not known what part of the overall footprint belongs to the tables in Rails. Hash tables might be a small part in comparison with other Rails data. Even if the tables are the

05/08/2021
lion part, their elements and keys might require more memory than table entries.

- Therefore it is hard to say what trade-off in performance/table size should be for Rails. I personally still prefer my April variant.

#142 - 11/04/2016 09:28 PM - vmakarov (Vladimir Makarov)

Vladimir Makarov wrote:

Koichi's investigation reported that small hash tables are majority during work of Ruby on Rails:

https://gist.github.com/ko1/dda8243134bc40f7c05a293abc54f2b2#file-report-md

Memory footprint of cloud applications like Ruby on Rails is important topic.

Since Koichi's report some changes have been done in my (https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing) and Yura's (https://github.com/funny-falcon/ruby/tree/st_table_with_array3) implementations.

Here are memory consumption graphs. The data produced by Koichi's script https://gist.githubusercontent.com/ko1/dda8243134bc40f7c05a293abc54f2b2/raw/dfccc2cc5e6f4f749c8a9390eae3104da84eb32f/stbench.rb on 4.2 GHz i7-4790K under FC24.

**size64bit.png**

On these graphs

- 'vlad current' means my current variant (https://github.com/vnmakarov/ruby.git, branch hash_tables_with_open_addressing, f4d2bf3ab78e59118e5dd89bccc5c3d8c4371ed35).
- 'vlad april' means my April variant without the last two commits to st.c.
- trunk means the current trunk (03c9bc2)
- yura means latest Yura's variant (https://github.com/funny-falcon/ruby.git, branch st_table_with_array3, 39d9b2c3658c2859be4b2edc1c7aeabahec6aaf) of tables with open addressing. I used a 64-bit variant, there is no practically difference in performance and memory consumption with 32-bit variant.

Here are the tables speed improvements relative to the trunk on MRI hash benchmarks:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Yura</th>
<th>My April</th>
<th>My current</th>
</tr>
</thead>
<tbody>
<tr>
<td>bighash</td>
<td>1.716</td>
<td>1.509</td>
<td>1.443</td>
</tr>
<tr>
<td>hash_aref_dsym</td>
<td>1.019</td>
<td>1.018</td>
<td>0.994</td>
</tr>
<tr>
<td>hash_aref_dsym_l</td>
<td>1.394</td>
<td>1.414</td>
<td>1.377</td>
</tr>
<tr>
<td>hash_aref_fix</td>
<td>0.984</td>
<td>1.041</td>
<td>1.015</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.713</td>
<td>1.832</td>
<td>1.828</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.177</td>
<td>1.225</td>
<td>1.222</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>0.959</td>
<td>1.015</td>
<td>1.012</td>
</tr>
<tr>
<td>hash_aref_sym_l</td>
<td>1.047</td>
<td>1.066</td>
<td>1.042</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.165</td>
<td>1.095</td>
<td>1.087</td>
</tr>
<tr>
<td>hash_ident_flo</td>
<td>1.005</td>
<td>0.949</td>
<td>0.958</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>0.932</td>
<td>0.979</td>
<td>0.954</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>0.886</td>
<td>0.934</td>
<td>0.919</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>0.919</td>
<td>0.942</td>
<td>0.919</td>
</tr>
<tr>
<td>hash_keys</td>
<td>3.024</td>
<td>3.039</td>
<td>3.073</td>
</tr>
<tr>
<td>hash_long</td>
<td>0.751</td>
<td>1.483</td>
<td>1.486</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.363</td>
<td>1.371</td>
<td>1.346</td>
</tr>
<tr>
<td>hash_shift_u16</td>
<td>1.388</td>
<td>1.392</td>
<td>1.359</td>
</tr>
</tbody>
</table>
The data are obtained by running the following script on the same machine used exclusively for benchmarking:

```
ruby ../ruby/benchmark/driver.rb -p hash -r 3 -e trunk::trunk/miniruby -e yura::yura/miniruby -e yura::yura64/miniruby -e current::./miniruby 2>/dev/null|awk 'NF==4 && /hash/ {s1+=$2;s2+=$3;s3+=$4;n++;print} END{print s1/n, s2/n, s3/n}'
```

Conclusions:

- My current variant requires less memory than April one. The current variant is close to average memory consumption of Yura's one (see the integral, a square of areas under the curves).
- Yura's curve is more smooth because of more frequent and smaller rebuilding but probably it results in the speed decrease.
- There is trade-off in my implementation (and probably Yura's ones) in table speed and its size.
- More size reduction in my tables by increasing table fill rate (e.g. from 0.5 to 3/4) is not worth it as one bin size is 1/24 - 1/12 of one entry size for tables with less 2^{15} elements.
- Unfortunately, it is not known what part of the overall footprint belongs to the tables in Rails. Hash tables might be a small part in comparison with other Rails data. Even if the tables are the lion part, their elements and keys might require more memory than table entries.
- Therefore it is hard to say what trade-off in performance/table size should be for Rails. I personally still prefer my April variant.

---

#143 - 11/04/2016 09:30 PM - funny_falcon (Yura Sokolov)
- File st_table_array.mbox added

#144 - 11/04/2016 10:03 PM - funny_falcon (Yura Sokolov)
I've uploaded refined mbox patch: https://bugs.ruby-lang.org/attachments/download/6225/st_table_array.mbox
Corresponding github branch: https://github.com/funny-falcon/ruby/commits/st_table_with_array5

It almost the same version Vladimir presents results for in last message, but with couple of further simplifications/optimizations.

It is rebased on top of trunk, and commit history is cleaned.

I kept mentioning Vladimir as a co-author (probably, there should be more).

#145 - 11/05/2016 08:48 AM - funny_falcon (Yura Sokolov)
- File st_table_array2.mbox added

#146 - 11/05/2016 08:58 AM - funny_falcon (Yura Sokolov)
I've uploaded new version.
It has one-line "security" fix in murmur_finish that doesn't change performance or memory footprint.

https://bugs.ruby-lang.org/attachments/download/6226/st_table_array2.mbox

---

05/08/2021
Performance-wise, the difference between two candidates is so small. I'm having a hard time making a decision. But to merge it before 2.4, we need to make a decision now.

So I chose Vlad's version as a baseline since it is slightly simpler. Probably we can merge it to the soon-to-be-released 2.4rc.

Matz.

Yukihiro Matsumoto wrote:

Performance-wise, the difference between two candidates is so small. I'm having a hard time making a decision. But to merge it before 2.4, we need to make a decision now.

So I chose Vlad's version as a baseline since it is slightly simpler. Probably we can merge it to the soon-to-be-released 2.4rc.

Matz.

Thank you.

I learned a lot from working with Yura. He deserves credits too as our competition results in a better hash tables. Some of his ideas could be added as separate patches.

I'd like to thank Koichi and other people who tested and benchmarked our hash tables.

I've just updated my branch to fix its compilation by LLVM-3.8.

The full patch can be obtained by

git diff trunk..hash_tables_with_open_addressing

in my repository https://github.com/vnmakarov/ruby.git

Is it a small (but slightly slow) version?
I prefer it.

Although my version contains a lot of non st_table related code (improving and securing siphash and murmur), my st_table code is much shorter and simpler.

https://github.com/funny-falcon/ruby/blob/st_table_with_array5/st.c#L136-L1023
https://github.com/vnmakarov/ruby/blob/hash_tables_with_open_addressing/st.c#L301-L1616

My version has 519 meaningful lines of code (ie removing comments and single braces), and Vlad's version has 731 meaningful LOC.

Why Vlad's version is simpler?

If I remove most of non st_table related changes today, will you reconsider to give second look to my version?

Excuse me again for this.
With regards,
Yura

#151 - 11/05/2016 04:48 PM - vmakarov (Vladimir Makarov)
Koichi Sasada wrote:

in my repository https://github.com/vnmakarov/ruby.git

Is it a small (but slightly slow) version?
I prefer it.

Yes. The head of the branch has smaller memory footprint for tables than my April version. And it is a bit slower than April version.

#152 - 11/05/2016 04:58 PM - funny_falcon (Yura Sokolov)

- File st_table_array4.mbox added

#153 - 11/05/2016 05:03 PM - funny_falcon (Yura Sokolov)

Sorry again for sending a version after accepted decision.
It is my last attempt to jump in a train:
https://bugs.ruby-lang.org/attachments/download/6228/st_table_array4.mbox

I removed all siphash related code. (but I kept murmur fixes and seeding).
Also it contains couple of improvements over the last version (LIKELY in st_foreach and a bit simpler add_direct).

Benchmark results for this version:

Speedup ratio: compare with the result of `trunk' (greater is better)

name    falcon    makarov
bighash 1.716  1.612
hash_aref_dsym 1.016  0.978
hash_aref_dsym_long 1.439  1.533
hash_aref_fix  0.975  0.987
hash_aref_flo   1.737  1.781
hash_aref_miss  1.141  1.097
hash_aref_str  1.226  1.271
hash_aref_sym  0.978  1.019
hash_aref_sym_long 1.058  1.049
hash_flatten  1.158  1.159
hash_ident_flo  0.994  0.917
hash_ident_num  0.917  0.912
hash_ident_obj  0.895  0.905
hash_ident_str  0.891  0.921
hash_ident_sym  0.974  0.964
hash_keys      2.817  2.841
hash_long      1.436  1.473
hash_shift     1.454  1.387
hash_shift_u16 1.420  1.387
hash_shift_u32 1.410  1.336
hash_small2    1.016  1.038
hash_small4    1.028  1.027
hash_small7    1.757  1.770
hash_small8    1.697  1.986
hash_to_proc   1.013  1.012
hash_values    2.802  2.821
loop_whileloop2 1.013  1.008
vm2_bighash*   3.060  3.056

github branch:
https://github.com/funny-falcon/ruby/tree/st_table_with_array6

#154 - 11/05/2016 09:16 PM - vmakarov (Vladimir Makarov)

Yura Sokolov wrote:

my st_table code is much shorter and simpler.

https://github.com/funny-falcon/ruby/blob/st_table_with_array5/st.c#L136-L1023
https://github.com/vnmakarov/ruby/blob/hash_tables_with_open_addressing/st.c#L301-L1616

05/08/2021
My version has 519 meaningful lines of code (ie removing comments and single braces), and Vlad's version has 731 meaningful LOC.

Why Vlad's version is simpler?

I think measuring lines after removing comments and single braces to define what is simpler is not an accurate approach:

- I suspect you did not count tables defining hash tables growth. In my variant, it is in st.c. In yours, it is in files st*.inc. And that is about 80 lines alone.
- I have more sophisticated function checking hash table consistency. It proved to be very useful in my debugging. My variant of the function is 30 lines longer.

In my variant, the call looks like:

```c
#ifdef ST_DEBUG
    st_check(tab);
#endif
```

In yours, you add non-debug variant (empty macro) and the call looks like:

```c
st_check(tab);
```

And it adds about 40 lines to my code. I think it is a matter of taste. I prefer to have possibility to switch on checking in one particular place.

Also I have 30 more asserts than you. I use it frequently also for documentation of expected conditions.

- Your code has
  ```c
  if (values >= values_end) break;
  ```
  mine always only
  ```c
  if (values == values_end)
    break;
  ```

- I have tendency to use longer identifiers (I found them useful for long maintenance as their names are a part of documentation. IMHO choosing the right names is very important part of coding).

Using longer names means sometimes longer lines which should be split.

- Using more functions called once helps code understanding in many cases but it increases # of lines.

I could continue but I stop. So in my opinion your definition of simplicity is very inaccurate.

#155 - 11/06/2016 01:35 AM - vmakarov (Vladimir Makarov)

Vladimir Makarov wrote:

- I suspect you did not count tables defining hash tables growth. In my variant, it is in st.c. In yours, it is in files st*.inc. And that is about 80 lines alone.

Sorry, my suspicion was wrong. It seems you excluded this in your line counting.

#156 - 11/06/2016 06:14 AM - funny_falcon (Yura Sokolov)

Vladimir,

Yes, I excluded tables, and I measured after preprocessor, so all #if are removed:

- removed function that checks table consistency (I also has such function, and it is also removed)
- removed all calls to such function and assertions Here is result after removing single 'break;' and function definition types (and I ought to redefine NULL and assert, cause they contained newlines and your st_assert, cause mine is no-op):
392 lines vs 546 lines. Is it fair now?

#157 - 11/06/2016 06:32 AM - funny_falcon (Yura Sokolov)

looks like I removed function definitions in previous count. fixed that.
now counts are: 428 lines vs 572 lines.

#158 - 11/06/2016 06:37 AM - funny_falcon (Yura Sokolov)

Any way,
I apologize again for my messages after decision.
It were inappropriate.
It is just emotional reaction.
I was ready to loose, but I really thought my version is simpler, so it is reaction only on that remark.

#159 - 11/07/2016 12:45 AM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset r56650.

Introduce table improvement by Vladimir Makarov vmakarov@redhat.com.
[Feature #12142]
See header of st.c for improvment details.

You can see all of code history here:
https://github.com/vnmakarov/ruby/tree/hash_tables_with_open_addressing

This improvement is discussed at
https://bugs.ruby-lang.org/issues/12142
with many people, especially with Yura Sokolov.

* st.c: improve st_table.
* include/ruby/st.h: ditto.
* internal.h, numeric.c, hash.c (rb_dbl_long_hash): extract a function.
* ext/-test-/st/foreach/foreach.c: catch up this change.

#160 - 11/07/2016 01:51 AM - duerst (Martin Dürst)

On 2016/11/06 15:37, funny_falcon@gmail.com wrote:

Issue #12142 has been updated by Yura Sokolov.

I was ready to lose,

Open Source Software is not a zero-sum game. I think that with the
patches now in Ruby trunk, everybody wins. You win, Vladimir wins,
everybody else wins, and Ruby wins! Thanks again to everybody.

but I really thought my version is simpler, so it is reaction only on that remark.

Software simplicity is in the eye of the beholder.

Some people (hopefully nobody on this list) will think that

puts 'Hello Ruby!'  
puts 'Hello Ruby!'  
puts 'Hello Ruby!'  
puts 'Hello Ruby!'  
puts 'Hello Ruby!'  

is simpler than

5.times { puts 'Hello Ruby!' }

Next time we get a chance, we can ask Matz why he thought that one
version was simpler than the other. But that may take some time; Matz is
very busy.
Regards, Martin.

#161 - 11/07/2016 03:47 AM - matz (Yukihiro Matsumoto)
Funny_falcon,

Sorry for not choosing yours. Both are as good, the difference is so slight, and I had to make a decision at the moment. I felt Vlad's version has some root for improvement that we see in yours. That's the reason I said "baseline".

Matz.

#162 - 11/07/2016 06:09 AM - funny_falcon (Yura Sokolov)
Matz,

I was disappointed not because you chose Vlad's version, but because you called it "simpler" than mine.
I'm glad that Ruby will have new implementation.
I do respect your choice.
I do respect Vladimir's work. I learned a lot from it.

With regards,
Yura

#163 - 11/07/2016 08:33 AM - ko1 (Koichi Sasada)
I updated my report:
https://gist.github.com/ko1/dda8243134bc40f7fc05e293abc54b2#file-report-md

- add vlad2 as last vlad's implementation
- add insertion overhead

Conclusion is not changed. Both implementations are faster than original st_table, but no huge difference between them. So I want to appreciate Matz's decision, and Of course, two of you huge efforts.

#164 - 11/07/2016 08:53 AM - funny_falcon (Yura Sokolov)
Koichi,

You've measured latest Vlad's version, but not latest mine.

Though, it doesn't matter now.

Regards,
Yura.

#165 - 11/07/2016 09:11 AM - ko1 (Koichi Sasada)
On 2016/11/07 17:53, funny_falcon@gmail.com wrote:

You've measured latest Vlad's version, but not latest mine.

There are several reasons:

(1) Your last version is not open addressing.
It is hard to compare many evaluation points.
This was main reason.

(2) You mentioned that chained version will increase memory usage.
At this time (before Ruby 2.4), we need to decide
maybe this is last chance to merge,
so I decide to see memory usage more for Ruby 2.4.

(3) Your patch is not on github, but with an attached patch.
It is slightly tough for me.
(I made a script to build ruby on svn/git branch)

(4) I measured them last week and I have no plenty time.
(this is why I post this update version today)

(3) and (4) is not fair reason. Only because my laziness (if I had shorten my sleeping time, I could do).

I hope my report will help next improvements.
Koichi,

1) My last version is open addressing
2) ... (so it doesn't use more memory)
3) It is on github
   https://github.com/funny-falcon/ruby/tree/st_table_with_array6
   (and I've posted link in https://bugs.ruby-lang.org/issues/12142#note-153)

Regards,
Yura

I understand, it was my mistake to push into new branch instead of st_table_with_array3.

On 2016/11/07 18:32, funny.falcon@gmail.com wrote:

1) My last version is open addressing
2) ... (so it doesn't use more memory)
3) It is on github
   https://github.com/funny-falcon/ruby/tree/st_table_with_array6
   (and I've posted link in https://bugs.ruby-lang.org/issues/12142#note-153)

Sorry, I couldn't check them (6 url).
As I wrote, I measured them in last week (Sat).

Yes, it was my mistake.
Thank you for all your efforts.

Regards,
Yura

I've rebased latest version on top of trunk, and measured it.

Here is result: https://gist.github.com/funny-falcon/956a49e77080d5a6d387dfd90d6d06d2

Regards.

Related to Bug #13002: Hash calculations no longer using universal hashing added

I think the array features should be static const.
https://github.com/k-takata/Onigmo/commit/44e3c0a16da1116be641ea807c1202434b743ace

Description updated
Subject changed from Forex reserves china to Hash tables with open addressing
- Project changed from CommonRuby to Ruby master

#175 - 12/25/2020 02:03 AM - Forexscenty (Forexscenty Forexscenty)
- Description updated
- Subject changed from Hash tables with open addressing to Best free forex demo account
- Project changed from Ruby master to CommonRuby

#176 - 12/25/2020 02:37 AM - nobu (Nobuyoshi Nakada)
- Subject changed from Best free forex demo account to Hash tables with open addressing
- Project changed from CommonRuby to Ruby master

#177 - 12/25/2020 02:39 AM - nobu (Nobuyoshi Nakada)
- Description updated

Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>st-march31.patch</td>
<td>114 KB</td>
<td>04/01/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>base.patch</td>
<td>93.8 KB</td>
<td>04/29/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>hash.patch</td>
<td>4.48 KB</td>
<td>04/29/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>strong_hash.patch</td>
<td>8.08 KB</td>
<td>04/29/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>city.patch</td>
<td>19.4 KB</td>
<td>04/29/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>new-hash-table-benchmarks.patch</td>
<td>1.34 KB</td>
<td>04/30/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>size16.png</td>
<td>6.91 KB</td>
<td>11/04/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>size256.png</td>
<td>6.95 KB</td>
<td>11/04/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>size60000.png</td>
<td>7.59 KB</td>
<td>11/04/2016</td>
<td>vmakarov (Vladimir Makarov)</td>
</tr>
<tr>
<td>st_table_array.mbox</td>
<td>102 KB</td>
<td>11/04/2016</td>
<td>funny_falcon (Yura Sokolov)</td>
</tr>
<tr>
<td>st_table_array2.mbox</td>
<td>102 KB</td>
<td>11/05/2016</td>
<td>funny_falcon (Yura Sokolov)</td>
</tr>
<tr>
<td>st_table_array4.mbox</td>
<td>89.5 KB</td>
<td>11/05/2016</td>
<td>funny_falcon (Yura Sokolov)</td>
</tr>
</tbody>
</table>