Ruby - Feature #21268

Implement a lock-free hash set for fstring table

04/16/2025 06:47 AM - jhawthorn (John Hawthorn)

Status:	Closed	
Priority:	Normal	
Assignee:	jhawthorn (John Hawthorn)	
Target version:	3.5	

Description

I would like to propose replacing the st_table used for fstrings (de-duplicated frozen strings) with a lock-free hash set to have improved performance in a Ractor environment.

Fstrings are shared between Ractors, and can be looked up concurrently by multiple Ractors, it's desirable to be able to do this work in parallel. There should be no change visible to users (other than improved performance :))

Previously this table used the VM lock (RB_VM_LOCK_ENTER, not the GVL) to serialize access. I first experimented with using a reader-writer lock, but that was a little awkward to use as it was easy to cause a deadlock between the new lock, RB_VM_LOCK_ENTER, and rb_vm_barrier(). An RW lock also can cause cache ping ponging (though striping can help). Lastly, this table is somewhat write-heavy, so it's desirable for even writes to be possible in parallel. A lock free table is not necessarily faster than a table with locking, but due to these constraints and usage I believe a lock-free table is desirable here.

The design is a fairly standard open-addressing hash table with quadratic probing. Similar to our existing st_table or id_table. Deletes (which happen on GC) replace existing keys with a tombstone, which is the only type of update which can occur. Tombstones are only cleared out on rebuilds (necessary for the lock-free design I used).

The set is wait-free for lookup (the thread performing lookup always makes progress) and lock-free for insert (we may sometimes retry if another thread made a concurrent insertion, we never spin on a lock) unless resizing/rebuilding.

The implementation itself is at https://github.com/ruby/ruby/pull/12921 and details are within. I will include benchmarks in a comment below.

Associated revisions

Revision 57b6a7503f19a9ffb53b1c505c7e093d7a6e33a4 - 04/18/2025 04:03 AM - jhawthorn (John Hawthorn)

Lock-free hash set for fstrings [Feature #21268]

This implements a hash set which is wait-free for lookup and lock-free
for insert (unless resizing) to use for fstring de-duplication.

As highlighted in https://bugs.ruby-lang.org/issues/19288, heavy use of
fstrings (frozen interned strings) can significantly reduce the
parallelism of Ractors.

I tried a few other approaches first: using an RWLock, striping a series
of RWlocks (partitioning the hash N-ways to reduce lock contention), and
putting a cache in front of it. All of these improved the situation, but
were unsatisfying as all still required locks for writes (and granular
locks are awkward, since we run the risk of needing to reach a vm
barrier) and this table is somewhat write-heavy.

My main reference for this was Cliff Click's talk on a lock free
hash-table for java https://www.youtube.com/watch?v=HJ-719EGIts. It
turns out this lock-free hash set is made easier to implement by a few
properties:

- We only need a hash set rather than a hash table (we only need keys, not values), and so the full entry can be written as a single VALUE
- As a set we only need lookup/insert/delete, no update
- Delete is only run inside GC so does not need to be atomic (It could be made concurrent)
- I use rb_vm_barrier for the (rare) table rebuilds (It could be made concurrent) We VM lock (but don't require other threads to stop) for table rebuilds, as those are rare
- The conservative garbage collector makes deferred replication easy, using a T_DATA object

Another benefits of having a table specific to fstrings is that we
compare by value on lookup/insert, but by identity on delete, as we only
want to remove the exact string which is being freed. This is faster and
provides a second way to avoid the race condition in
https://bugs.ruby-lang.org/issues/21172.

This is a pretty standard open-addressing hash table with quadratic
probing. Similar to our existing st_table or id_table. Deletes (which
happen on GC) replace existing keys with a tombstone, which is the only
type of update which can occur. Tombstones are only cleared out on
resize.

Unlike st_table, the VALUEs are stored in the hash table itself
(st_table's bins) rather than as a compact index. This avoids an extra
pointer dereference and is possible because we don't need to preserve
insertion order. The table targets a load factor of 2 (it is enlarged
once it is half full).

Revision 57b6a7503f19a9ffb53b1c505c7e093d7a6e33a4 - 04/18/2025 04:03 AM - jhawthorn (John Hawthorn)

Lock-free hash set for fstrings [Feature #21268]

This implements a hash set which is wait-free for lookup and lock-free
for insert (unless resizing) to use for fstring de-duplication.

As highlighted in https://bugs.ruby-lang.org/issues/19288, heavy use of
fstrings (frozen interned strings) can significantly reduce the
parallelism of Ractors.

I tried a few other approaches first: using an RWLock, striping a series
of RWlocks (partitioning the hash N-ways to reduce lock contention), and
putting a cache in front of it. All of these improved the situation, but
were unsatisfying as all still required locks for writes (and granular
locks are awkward, since we run the risk of needing to reach a vm
barrier) and this table is somewhat write-heavy.

My main reference for this was Cliff Click's talk on a lock free
hash-table for java https://www.youtube.com/watch?v=HJ-719EGIts. It
turns out this lock-free hash set is made easier to implement by a few
properties:

- We only need a hash set rather than a hash table (we only need keys,
  not values), and so the full entry can be written as a single VALUE
- As a set we only need lookup/insert/delete, no update
- Delete is only run inside GC so does not need to be atomic (It could
  be made concurrent)
- I use rb_vm_barrier for the (rare) table rebuilds (It could be made
  concurrent) We VM lock (but don't require other threads to stop) for
  table rebuilds, as those are rare
- The conservative garbage collector makes deferred replication easy,
  using a T_DATA object

Another benefits of having a table specific to fstrings is that we
compare by value on lookup/insert, but by identity on delete, as we only
want to remove the exact string which is being freed. This is faster and
provides a second way to avoid the race condition in
https://bugs.ruby-lang.org/issues/21172.

This is a pretty standard open-addressing hash table with quadratic
probing. Similar to our existing st_table or id_table. Deletes (which
happen on GC) replace existing keys with a tombstone, which is the only
type of update which can occur. Tombstones are only cleared out on
resize.

Unlike st_table, the VALUEs are stored in the hash table itself
(st_table's bins) rather than as a compact index. This avoids an extra
pointer dereference and is possible because we don't need to preserve
insertion order. The table targets a load factor of 2 (it is enlarged
once it is half full).

Revision 57b6a750 - 04/18/2025 04:03 AM - jhawthorn (John Hawthorn)

Lock-free hash set for fstrings [Feature #21268]

This implements a hash set which is wait-free for lookup and lock-free for insert (unless resizing) to use for fstring de-duplication.

As highlighted in https://bugs.ruby-lang.org/issues/19288, heavy use of fstrings (frozen interned strings) can significantly reduce the parallelism of Ractors.

I tried a few other approaches first: using an RWLock, striping a series of RWlocks (partitioning the hash N-ways to reduce lock contention), and putting a cache in front of it. All of these improved the situation, but were unsatisfying as all still required locks for writes (and granular locks are awkward, since we run the risk of needing to reach a vm barrier) and this table is somewhat write-heavy.

My main reference for this was Cliff Click's talk on a lock free hash-table for java https://www.youtube.com/watch?v=HJ-719EGIts. It turns out this lock-free hash set is made easier to implement by a few properties:

- We only need a hash set rather than a hash table (we only need keys, not values), and so the full entry can be written as a single VALUE
- As a set we only need lookup/insert/delete, no update
- Delete is only run inside GC so does not need to be atomic (It could be made concurrent)
- I use rb_vm_barrier for the (rare) table rebuilds (It could be made concurrent) We VM lock (but don't require other threads to stop) for table rebuilds, as those are rare
- The conservative garbage collector makes deferred replication easy, using a T_DATA object

Another benefits of having a table specific to fstrings is that we compare by value on lookup/insert, but by identity on delete, as we only want to remove the exact string which is being freed. This is faster and provides a second way to avoid the race condition in https://bugs.ruby-lang.org/issues/21172.

This is a pretty standard open-addressing hash table with quadratic probing. Similar to our existing st_table or id_table. Deletes (which happen on GC) replace existing keys with a tombstone, which is the only type of update which can occur. Tombstones are only cleared out on resize.

Unlike st_table, the VALUEs are stored in the hash table itself (st_table's bins) rather than as a compact index. This avoids an extra pointer dereference and is possible because we don't need to preserve insertion order. The table targets a load factor of 2 (it is enlarged once it is half full).

History

#1 - 04/17/2025 01:38 AM - jhawthorn (John Hawthorn)

I've added two new benchmarks to measure this. This is on my M2 macbook air. When run in Ractors inserting is ~3x faster and lookups are ~8x faster. In a single Ractor performance is about the same.

```
compare-ruby: ruby 3.5.0dev (2025-04-15T23:29:23Z master 2cf95e2e04) +PRISM [arm64-darwin24]
built-ruby: ruby 3.5.0dev (2025-04-16T03:10:21Z ractor_fstring_has.. 20acd942ae) +PRISM [arm64-darwin24]
# Iteration per second (i/s)

|                       |compare-ruby|built-ruby|
|:----------------------|-----------:|---------:|
|ractor_fstring_random  |       0.567|     1.671|
|                       |           -|     2.95x|
|ractor_fstring_same    |       1.132|     9.917|
|                       |           -|     8.76x|
# Iteration per second (i/s)

|                |compare-ruby|built-ruby|
|:---------------|-----------:|---------:|
|fstring_random  |       1.499|     1.383|
|                |       1.08x|         -|
|fstring_same    |       2.141|     2.254|
|                |           -|     1.05x|
```

#2 - 04/18/2025 04:04 AM - jhawthorn (John Hawthorn)

- Status changed from Open to Closed

Applied in changeset git|57b6a7503f19a9ffb53b1c505c7e093d7a6e33a4.

Lock-free hash set for fstrings [Feature #21268]

This implements a hash set which is wait-free for lookup and lock-free
for insert (unless resizing) to use for fstring de-duplication.

As highlighted in https://bugs.ruby-lang.org/issues/19288, heavy use of
fstrings (frozen interned strings) can significantly reduce the
parallelism of Ractors.

I tried a few other approaches first: using an RWLock, striping a series
of RWlocks (partitioning the hash N-ways to reduce lock contention), and
putting a cache in front of it. All of these improved the situation, but
were unsatisfying as all still required locks for writes (and granular
locks are awkward, since we run the risk of needing to reach a vm
barrier) and this table is somewhat write-heavy.

My main reference for this was Cliff Click's talk on a lock free
hash-table for java https://www.youtube.com/watch?v=HJ-719EGIts. It
turns out this lock-free hash set is made easier to implement by a few
properties:

- We only need a hash set rather than a hash table (we only need keys, not values), and so the full entry can be written as a single VALUE
- As a set we only need lookup/insert/delete, no update
- Delete is only run inside GC so does not need to be atomic (It could be made concurrent)
- I use rb_vm_barrier for the (rare) table rebuilds (It could be made concurrent) We VM lock (but don't require other threads to stop) for table rebuilds, as those are rare
- The conservative garbage collector makes deferred replication easy, using a T_DATA object

Another benefits of having a table specific to fstrings is that we
compare by value on lookup/insert, but by identity on delete, as we only
want to remove the exact string which is being freed. This is faster and
provides a second way to avoid the race condition in
https://bugs.ruby-lang.org/issues/21172.

This is a pretty standard open-addressing hash table with quadratic
probing. Similar to our existing st_table or id_table. Deletes (which
happen on GC) replace existing keys with a tombstone, which is the only
type of update which can occur. Tombstones are only cleared out on
resize.

Unlike st_table, the VALUEs are stored in the hash table itself
(st_table's bins) rather than as a compact index. This avoids an extra
pointer dereference and is possible because we don't need to preserve
insertion order. The table targets a load factor of 2 (it is enlarged
once it is half full).