Ruby trunk - Feature #14989

Add Hash support for transient heap

08/14/2018 03:38 AM - tacinight (Yimin Zhao)

Status:	Closed
Priority:	Normal
Assignee:	ko1 (Koichi Sasada)
Target version:	

Description

Hi, there
I am a GSoC student following ko1 (Koichi Sasada) to do some work for Ruby, here I want to make a brief introduction for one of my work in this period.

Transient Heap

The first discussion of transient heap was opened by ko1 (Koichi Sasada) at Introduce 2nd GC heap named Transient heap. In his prototype, he completed the implement for arrays, and the future work should also support the type String, Hash as well as Extend(shrink) policy for transient heap.

My work here is adding support for Hash so that transient heap can allocate space for Hash. I encounted some difficults that had to be handled.

1. Hash type objects use st_table to store elements, the correlation between those two is weak. For transient heap, it has to know the ruby type of the objects.
2. Transient heap is based on the hypothesis that most of objects die at a young age. To understand the issue with first need to survey the distribution of hash size.

The distribution of hash size

Implement

The capacity of st_table is always a power of 2, we recorded the exponent into a global array before gc or functions explicitly frees st_table. Look at the patch for more details.

Results

We measured 6 applications in order to find a real distribution of the hash size. Respectively, those applications are:

1. make check command under ruby source code.
2. make gcbench_rdoc command under ruby source code.
3. gem uninstall --all to uninstall over 300 gems
4. bundle install commmand under project discourse
5. discourse manually explore the web pages and simulate the different events for about 20 minutes (development environment)
6. rails_ruby_bench project rails_ruby_bench with parameters : 8 workers, 1500 iterations
7. rails_ruby_bench 2 8 workers, 12000 iterations
8. rails_ruby_bench 3 20 workders, 20000 iterations

The results show that objects under size 8(area of exponent 2 to 3) account for more than 80% of the total data in most situations.
The raw data can be found at st_table size statistics.

Add Hash support

According to the previous survey result, we use a array-like fixed-size table to store the elements. We name it as LinearTable.
In this way, we have two benefits:

1. We don't have to create st_table for small hash(but it will when necessary)
2. the code is written into hash.c (while st_table code is in st.c). its easy to bind small hash with transient heap, and by default, when using st_table, it uses the malloced space.

So the basic data structure look like this.

```
#define LINEAR_TABLE_MAX_SIZE 8
```

```
typedef struct li_table_entry {
    VALUE hash;
    VALUE key;
    VALUE record;
} li_table_entry;

typedef struct LinearTable {
    const struct st_hash_type *type;
    li_table_entry entries[LINEAR_TABLE_MAX_SIZE];
} li_table;

struct RHash {
    struct RBasic basic;
    union {
    struct st_table *ntbl;      /* possibly 0 */
    struct LinearTable *ltbl;
    } as;
    int iter_lev;
    const VALUE ifnone;
};
```

The layout of li_table_entry

Considering the size of 8 hashes/keys/records is just the size of a cache line, I also tried different data layouts.

```
#define LINEAR_TABLE_MAX_SIZE 8

// the original layout
typedef struct li_table_entry {
    VALUE hash;
    VALUE key;
    VALUE record;
} li_table_entry;

typedef struct LinearTable {
    const struct st_hash_type *type;
    li_table_entry entries[LINEAR_TABLE_MAX_SIZE];
} li_table;
```

```
#define LINEAR_TABLE_MAX_SIZE 8

// the original layout
typedef struct li_table_entry {
    VALUE hash;
    VALUE key;
    VALUE record;
} li_table_entry;

typedef struct LinearTable {
    const struct st_hash_type *type;
    li_table_entry entries[LINEAR_TABLE_MAX_SIZE];
} li_table;
```

```
#define LINEAR_TABLE_MAX_SIZE 8

// The Variant 1
typedef struct li_table_entry {
    VALUE key;
    VALUE record;
} li_table_entry;

typedef struct LinearTable {
    const struct st_hash_type *type;
    VALUE hashes[LINEAR_TABLE_MAX_SIZE];
    li_table_entry pairs[LINEAR_TABLE_MAX_SIZE];
} li_table;
```

```
#define LINEAR_TABLE_MAX_SIZE 8

// The Variant 2
typedef struct li_table_entry {
    VALUE hash;
    VALUE key;
} li_table_entry;

typedef struct LinearTable {
    const struct st_hash_type *type;
    li_table_entry entries[LINEAR_TABLE_MAX_SIZE];
    VALUE records[LINEAR_TABLE_MAX_SIZE];
} li_table;

#define LINEAR_TABLE_MAX_SIZE 8

// The Variant 3
typedef struct LinearTable {
    const struct st_hash_type *type;
    VALUE hashes[LINEAR_TABLE_MAX_SIZE];
    VALUE keys[LINEAR_TABLE_MAX_SIZE];
    VALUE records[LINEAR_TABLE_MAX_SIZE];
} li_table;
```

The microbench results show the original edition and variant 1 have better performance than the other two. The patches can be found in the repository as linear_table_v1.patch, linear_table_v2.patch, linear_table_v3.patch, linear_table_v4.patch;

Integrate data into flag bits

Considering the maximum size of entries is 8. We can store the num_entries and num_bound into RBasic(hash)->flag;
Related patch is here which based on linear_table_v1.patch

Benchmark results

I execuated comprehensive benchmark test based on official benchmark tool - benchmark-driver.

1. linear_table_benchmark, includes the make benchmark results of trunk vs. v1(the original version) vs. v2(the variant 1).
2. transient hash benchmark, includes the make benchmark results of trunk vs. transient_heap(based on v1) vs. integrated flag
3. linear_table_variant_benchmark, includes the micro-benchmark results of trunk vs. v1 vs. v2 vs. v3 vs. v4. The used benchrmark scripts are located at microbench directory.
4. linear_table real app bench, includes the test results of the six applications noted above.
5. transient heap microbench, include the micro-benchrmark resutls of trunk vs. v1 vs. transient-hash vs integrated-flag. This time I also add memory usage comparison.
6. linear_table_vary_size_microbench, we once worried the size of linear table will impact on the benchmark results because of the linear search. So we test the linear table with different size and the results show the table size make no relevant difference.

Future work

1. In transient hash benchmark, we saw a preformance degradation compare to the preformance improvement in linear_table_benchmark in same items. It needs to be analyzed and to be solved.

My GitHub Repository

I will update my status at a GitHub repository: https://github.com/Tacinight/ruby-gsoc-2018.
More explanatory images and other of my work also can be found at this repository.
Welcome to review the patches and leave your valuable comments.

Associated revisions

Revision 8f675cdd - 10/30/2018 10:11 PM - ko1 (Koichi Sasada)

support theap for T_HASH. [Feature #14989]

- hash.c, internal.h: support theap for small Hash. Introduce RHASH_ARRAY (li_table) besides st_table and small Hash (<=8 entries) are managed by an array data structure. This array data can be managed by theap. If st_table is needed, then converting array data to st_table data.

For st_table using code, we prepare "stlike" APIs which accepts hash value
and are very similar to st_ APIs.

This work is based on the GSoC achievement
by tacinight tacingiht@gmail.com and refined by ko1.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65454 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 65454 - 10/30/2018 10:11 PM - ko1 (Koichi Sasada)

support theap for T_HASH. [Feature #14989]

- hash.c, internal.h: support theap for small Hash. Introduce RHASH_ARRAY (li_table) besides st_table and small Hash (<=8 entries) are managed by an array data structure. This array data can be managed by theap. If st_table is needed, then converting array data to st_table data.

For st_table using code, we prepare "stlike" APIs which accepts hash value
and are very similar to st_ APIs.

This work is based on the GSoC achievement
by tacinight tacingiht@gmail.com and refined by ko1.

Revision 65454 - 10/30/2018 10:11 PM - ko1 (Koichi Sasada)

support theap for T_HASH. [Feature #14989]

- hash.c, internal.h: support theap for small Hash. Introduce RHASH_ARRAY (li_table) besides st_table and small Hash (<=8 entries) are managed by an array data structure. This array data can be managed by theap. If st_table is needed, then converting array data to st_table data.

For st_table using code, we prepare "stlike" APIs which accepts hash value
and are very similar to st_ APIs.

This work is based on the GSoC achievement
by tacinight tacingiht@gmail.com and refined by ko1.

History

#1 - 08/14/2018 04:33 AM - shyouhei (Shyouhei Urabe)

Thank you for a great work.

tacinight (Yimin Zhao) wrote:

> 1. Transient heap is based on the hypothesis that most of objects die at a young age. To understand the issue with first need to survey the distribution of hash size.

Why you can say that a hash's size has something to do with its age?

> ### Add Hash support
>
> According to the previous survey result, we use a array-like fixed-size table to store the elements. We name it as LinearTable.
> In this way, we have two benefits:
>
> 1. We don't have to create st_table for small hash(but it will when necessary)

As you should have been aware, we already do something very similar to struct LinearTable (see MAX_POWER2_FOR_TABLES_WITHOUT_BINS if you have missed). Is this really a benefit?

> ### My GitHub Repository
>
> I will update my status at a GitHub repository: https://github.com/Tacinight/ruby-gsoc-2018.
> More explanatory images and other of my work also can be found at this repository.
> Welcome to review the patches and leave your valuable comments.

I find it difficult to navigate the repository. Where can I find the exact patch that we should look at? Is it
0002-add-transient-heap-linear-table-support.patch ?

#2 - 08/14/2018 07:32 AM - tacinight (Yimin Zhao)

Thanks for the quick response.

> Why you can say that a hash's size has something to do with its age?

Thanks for pointing it out, I admit that there are some problems with
this statement. I will go to measure it.

1. We don't have to create st_table for small hash(but it will when necessary)

> As you should have been aware, we already do something very similar to struct LinearTable (see MAX_POWER2_FOR_TABLES_WITHOUT_BINS if you have missed). Is this really a benefit?

Yes, I know it.
It is not about a optimization for st_table, the question is that
transient heap has to know the obj builtin type,
like T_ARRAY, or T_HASH, LinearTable serves as a temporary space to
store small hash objs which
request space from transient heap.
st_table is more like a data structure with general purposes widely
used in internal ruby. It will need massive
rewriting if bind it with hash.

> I find it difficult to navigate the repository. Where can I find the exact patch that we should look at? Is it 0002-add-transient-heap-linear-table-support.patch ?

Sorry for the inconvenience, I merged separate patches into one, you
can go to check this one:
https://github.com/Tacinight/ruby-gsoc-2018/blob/master/patch/add-hash-support-for-transient-heap.patch

Thanks
Yimin

#3 - 08/14/2018 10:44 AM - shevegen (Robert A. Heiler)

Nice introduction and explanation!

#4 - 08/15/2018 01:36 AM - shyouhei (Shyouhei Urabe)

Hello, thank you again for a response.

tacinight (Yimin Zhao) wrote:

> > Why you can say that a hash's size has something to do with its age?
>
> Thanks for pointing it out, I admit that there are some problems with
> this statement. I will go to measure it.

OK. Looking forward to the measurement result. Even if this was a false assert, moving small Hashs into transient heap should not be a bad idea I think.

> > 1. We don't have to create st_table for small hash(but it will when necessary)
>
> > As you should have been aware, we already do something very similar to struct LinearTable (see MAX_POWER2_FOR_TABLES_WITHOUT_BINS if you have missed). Is this really a benefit?
>
> Yes, I know it.
> It is not about a optimization for st_table, the question is that
> transient heap has to know the obj builtin type,
> like T_ARRAY, or T_HASH, LinearTable serves as a temporary space to
> store small hash objs which
> request space from transient heap.
> st_table is more like a data structure with general purposes widely
> used in internal ruby. It will need massive
> rewriting if bind it with hash.

I agree that st_table is a general-purpose data structure.
I also agree that rewriting st.c requires a lot to do.
However, isn't it just a matter of allocating struct st_table in the transient heap?
Because when you encounter a Hash instance during GC, its RHASH(obj)->ntbl shall never be shared among others.
You don't have to worry about its generic usages.
Or am I missing something?

> > I find it difficult to navigate the repository. Where can I find the exact patch that we should look at? Is it 0002-add-transient-heap-linear-table-support.patch ?

> Sorry for the inconvenience, I merged separate patches into one, you
> can go to check this one:
> https://github.com/Tacinight/ruby-gsoc-2018/blob/master/patch/add-hash-support-for-transient-heap.patch

Thank you. Will take a look.

#5 - 08/23/2018 07:25 AM - ko1 (Koichi Sasada)

- Assignee set to ko1 (Koichi Sasada)

#6 - 08/24/2018 08:26 AM - shyouhei (Shyouhei Urabe)

```
Hello, I took a look at add-hash-support-for-transient-heap.patch.

It seems lines are copy & pasted from st.c to hash.c.  This is in fact
a wise idea to avoid common pitfalls.  I have almost no comments on
that part.

> --- a/array.c
> +++ b/array.c
> @@ -4379,11 +4379,12 @@ static inline void
>  ary_recycle_hash(VALUE hash)
>  {
>      assert(RBASIC_CLASS(hash) == 0);
> -    if (RHASH(hash)->ntbl) {
> -     st_table *tbl = RHASH(hash)->ntbl;
> +    if (RHASH_TABLE_P(hash)) {
> +     st_table *tbl = RHASH(hash)->as.ntbl;
>       st_free_table(tbl);
> +     RHASH(hash)->as.ntbl = NULL;
>      }
> -    rb_gc_force_recycle(hash);
> +    //rb_gc_force_recycle(hash);
>  }

1. This project don't use // -style comments.  Please follow our
   style.

2. I guess this part is incomplete?  Otherwise tell us why you
   comment out the call of rb_gc_force_recycle.

3. What if RHASH_ARRAY_P(hash) is true? Does that never happen
   here?

> @@ -4158,6 +4159,20 @@ mark_hash(rb_objspace_t *objspace, st_table *tbl)
>      st_foreach(tbl, mark_keyvalue, (st_data_t)objspace);
>  }
>
> +static void
> +mark_hash_linear(rb_objspace_t *objspace, VALUE hash)
> +{
> +    if (RHASH_ARRAY_P(hash)) {
> +        linear_foreach(hash, mark_keyvalue, (st_data_t)objspace);
> +       if (objspace->mark_func_data == NULL && RHASH_TRANSIENT_P(hash)) {
> +            rb_transient_heap_mark(hash, RHASH(hash)->as.ltbl);
> +        }
> +    }
> +    else if (RHASH_TABLE_P(hash))
> +        st_foreach(RHASH(hash)->as.ntbl, mark_keyvalue, (st_data_t)objspace);
> +    gc_mark(objspace, RHASH(hash)->ifnone);
> +}
> +
>  void
>  rb_mark_hash(st_table *tbl)
>  {

1. This indentation is broken.  Seems you mix tabs and spaces.  You
   might want to use a text editor that help you indent things.

2. I guess you also have to modify rb_mark_hash() to use this new
   routine.

> +#define SET_KEY(entry, _key) (entry)->key = (_key)
```

```
> +#define SET_HASH(entry, _hash) (entry)->hash = (_hash)
> +#define SET_RECORD(entry, _value) (entry)->record = (_value)
```

Are these SET_* macros actually necessary? I see no
reason why you don't directly assign them.

```
> +static li_table*
> +linear_init_table(VALUE hash, const struct st_hash_type *type)
> +{
> +    li_table *tab;
> +    uint8_t i;
> +    tab = (li_table*)rb_transient_heap_alloc(hash, sizeof(li_table));
> +    if (tab != NULL) {
> +       RHASH_SET_TRANSIENT_FLAG(hash);
> +    }
> +    else {
> +       RHASH_UNSET_TRANSIENT_FLAG(hash);
> +       tab = (li_table*)malloc(sizeof(li_table));
> +    }
```

So from this part it seems you are implementing linear table that
is NOT transient. Your patch have following types of hash
tables:

- Size 0; that allocates nothing.
- Linear table allocated in transient heap.
- Linear table allocated using malloc.
- Good old st_table backended hash.

I guess it makes rb_hash_heap_free() overly complicated. Is
there any reason to support this much variants?

```
> +static void
> +try_convert_table(VALUE hash)
> +{
```

I don't like the function name. It describes nothing. What does
it do? Try converting from what to what?

```
> +    new_tab = st_init_table_with_size(tab->type, size * 2);
```

Can I ask you the reason behind your choice of this magic number?

```
> +    for (i = 0; i < LINEAR_TABLE_BOUND; i++) {
> +       HASH_ASSERT(entries[i].hash != 0);
> +       st_add_direct(new_tab, entries[i].key, entries[i].record);
> +    }
```

It seems you forgot to treat empty entries.

```
> +static st_table *
> +force_convert_table(VALUE hash)
> +{
```

I don't like the idea that try_convert_table(x) and
force_convert_table(x) return different things.

```
> +static int
> +add_direct_with_hash(VALUE hash, st_data_t key, st_data_t val, st_hash_t hash_value)
> +{
```

I could not understand what this return value means.

```
> +static void
> +linear_clear(VALUE hash)
> +{
> +    li_table *tab = RHASH(hash)->as.ltbl;
> +    RHASH_SET_ARRAY_LEN(hash, 0);
> +    RHASH_SET_ARRAY_BOUND(hash, 0);
> +    memset(tab->entries, 0, LINEAR_TABLE_MAX_SIZE * sizeof(li_table_entry));
> +}
```

This is different from what you do in linear_init_table(). You should
take either way and use in both functions.

```
> @@ -330,7 +952,7 @@ st_foreach_safe(st_table *table, int (*func)(ANYARGS), st_data_t a)
>      arg.func = (st_foreach_func *)func;
>      arg.arg = a;
>      if (st_foreach_check(table, foreach_safe_i, (st_data_t)&arg, 0)) {
> -       rb_raise(rb_eRuntimeError, "hash modified during iteration");
> +       rb_raise(rb_eRuntimeError, "1hash modified during iteration");
>      }
>  }
```

I don't understand this part.

```
> +void rb_hash_free(VALUE);
```

I see this identical declaration duplicated in mutiple header files.
Pick one place and delete others.

#7 - 10/30/2018 01:15 PM - k0kubun (Takashi Kokubun)

Do we have non-micro benchmark results for this? Maybe gcbench-rdoc on [Bug #14858] is the one?

> rails_ruby_bench project rails_ruby_bench with parameters : 8 workers, 1500 iterations
> rails_ruby_bench 2 8 workers, 12000 iterations
> rails_ruby_bench 3 20 workders, 20000 iterations

These things are mentioned for investigating the distribution of hash size. I'm curious about the result of the benchmark after this change, since we would not be able to evaluate this by another macro benchmark project, Optcarrot.

#8 - 10/30/2018 10:12 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset trunk|r65454.

---

support theap for T_HASH. [Feature #14989]

- hash.c, internal.h: support theap for small Hash. Introduce RHASH_ARRAY (li_table) besides st_table and small Hash (<=8 entries) are managed by an array data structure. This array data can be managed by theap. If st_table is needed, then converting array data to st_table data.

For st_table using code, we prepare "stlike" APIs which accepts hash value
and are very similar to st_ APIs.

This work is based on the GSoC achievement
by tacinight tacingiht@gmail.com and refined by ko1.