Ruby master - Feature #14370

Directly mark instruction operands and avoid mark_ary usage on rb_iseq_constant_body

01/17/2018 08:12 PM - tenderlovemaking (Aaron Patterson)

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

Description

Hi,

I've attached a patch that changes rb_iseq_mark to directly mark instruction operands rather than adding them to a mark array. I observed a ~3% memory reduction by directly marking operands, and I didn't observe any difference in GC time. To test memory usage, I used a basic Rails application, logged all malloc / free calls to a file, then wrote a script that would sum the live memory at each sample (each sample being a call to malloc). I graphed these totals so that I could see the memory usage as malloc calls were made:

35020270-1b0ded20-fae0-11e7-9cbd-1d028a6c9484.png
The red line is trunk, the blue line is trunk + the patch I've attached. Since the X axis is sample number (not time), the blue line is not as long as the red line because the blue line calls malloc fewer times. The Y axis in the graph is the total number of "live" bytes that have been allocated (all allocations minus their corresponding frees). You can see from the graph that memory savings start adding up as more code gets loaded.

I was concerned that this patch might impact GC time, but make gcbench-rdoc didn't seem to show any significant difference in GC time between trunk and this patch. If it turns out there is a performance impact, I think I could improve the time while still keeping memory usage low by generating a bitmap during iseq compilation.

There is a bit more information where I've been working, but I think I've summarized everything here.

https://github.com/github/ruby/pull/39

Related issues:

Related to Ruby master - Bug #14596: Ruby master is broken with bootsnap	**Open**

History

#1 - 01/17/2018 11:32 PM - normalperson (Eric Wong)

tenderlove@ruby-lang.org wrote:

> I was concerned that this patch might impact GC time, but
> make gcbench-rdoc didn't seem to show any significant
> difference in GC time between trunk and this patch. If it
> turns out there is a performance impact, I think I could
> improve the time while still keeping memory usage low by
> generating a bitmap during iseq compilation.

I like this patch so far. This improves data locality (at the cost of extra branches), so I'm all for the space reduction.

Also, iseq aren't marked frequently anymore since RGenGC; so I don't believe a bitmap index would be worth the extra code and space; this is probably why GC time isn't as noticeable.

```
+static int
+iseq_extract_values(const VALUE *code, size_t pos, iseq_value_itr_t * func, void *data)
```

Did you try without using function pointers? It may be possible to eek out a few more cycles by calling rb_gc_mark directly.

```
+void
+rb_iseq_each_value(const rb_iseq_t *iseq, iseq_value_itr_t * func, void *data)
```

Shouldn't this be static? (or do you have other changes planned?)

> +static void
> +each_insn_value(void *ctx, VALUE obj)
> +{
>
> - return rb_gc_mark(obj); +}

Needless "return" statement. I remember some compilers/options complain about the "return" despite rb_gc_mark also being void, even.

Thanks!

#2 - 01/18/2018 06:53 PM - tenderlovemaking (Aaron Patterson)

- File iseq_mark.diff added

normalperson (Eric Wong) wrote:

> tenderlove@ruby-lang.org wrote:
>
> > I was concerned that this patch might impact GC time, but make gcbench-rdoc didn't seem to show any significant difference in GC time between trunk and this patch. If it turns out there is a performance impact, I think I could improve the time while still keeping memory usage low by generating a bitmap during iseq compilation.
>
> I like this patch so far. This improves data locality (at the cost of extra branches), so I'm all for the space reduction.
>
> Also, iseq aren't marked frequently anymore since RGenGC; so I don't believe a bitmap index would be worth the extra code and space; this is probably why GC time isn't as noticeable.

I figured this might be the case. This patch should make ISeq marking slower, but I wasn't sure if it would be mitigated by the fact that the ISeq objects get old. It seems like RGenGC solves the problem.

> > +static int
> > +iseq_extract_values(const VALUE *code, size_t pos, iseq_value_itr_t * func, void *data)
>
> Did you try without using function pointers? It may be possible to eek out a few more cycles by calling rb_gc_mark directly.

I did not. I've got another branch going with a compacting garbage collector, and I'm reusing this function for updating references. Since that branch isn't ready for merging, I'm happy to change this to direct calls if it helps to get the patch merged (though I'd prefer to keep the function pointer as it makes my diff smaller). :)

> > +void
> > +rb_iseq_each_value(const rb_iseq_t *iseq, iseq_value_itr_t * func, void *data)
>
> Shouldn't this be static? (or do you have other changes planned?)

Yep, it should be static. Thank you!

> > +static void
> > +each_insn_value(void *ctx, VALUE obj)
> > +{
> >
> > - return rb_gc_mark(obj); +}
>
> Needless "return" statement. I remember some compilers/options complain about the "return" despite rb_gc_mark also being void, even.

Removed!

Thanks for the review. I've uploaded a new patch with the static change, and removed the needless return statement.

Another benefit of this patch that I forgot to mention is that it makes reading heap dumps much easier. Before you have to chase references through an array: iseq->array->literal, but now it's just iseq->literal.

Anyway, thanks again for the review! :)

#3 - 01/19/2018 02:16 AM - ko1 (Koichi Sasada)

Cool.

Did you verify the references between your patch and current implementation?

You can have two sets of referring objects from iseq->mark_ary and iseq (w/ your patch) and you can compare them to verify the reference.

> I figured this might be the case. This patch should make ISeq marking slower, but I wasn't sure if it would be mitigated by the fact that the ISeq objects get old. It seems like RGenGC solves the problem.

As you two said, it is not problem I also think (hopefully). But I want to know "how slow".

Thanks,
Koichi

#4 - 01/20/2018 02:51 AM - tenderlovemaking (Aaron Patterson)

- File bench.rb added

- File benchmark_methods.diff added

- File iseq_mark.diff added

ko1 (Koichi Sasada) wrote:

> Cool.
>
> Did you verify the references between your patch and current implementation?
>
> You can have two sets of referring objects from iseq->mark_ary and iseq (w/ your patch) and you can compare them to verify the reference.

Yes. I had to add marking for catch table iseq as well as marking the instructions. To verify the array contents I used the iterator code from this patch and printed out addresses from the instructions, then I compared those addresses to the addresses inside the mark array. I added code to the GC to trace C stacks for allocations (I think I could add this to allocation_tracer gem):

https://gist.github.com/tenderlove/4e74aebeb308cfa00306a186a26c0b8d

Using stack information plus mark_ary address and disasm addresses, I was able to make them match.

Unfortunately I cannot eliminate the mark array because we store coverage information, flip flop info, as well as result of once instruction (one time compile regex) in the mark array:

https://github.com/ruby/ruby/blob/4b88d9c1c49b2ea33aa91995251b955aa8b90953/vm_insnhelper.c#L3292-L3293

I think "once" isn't very popular, so it's fine to allow the mark array grow for that. Also the current mark_ary is small enough to be embedded, so I don't think there is a reason to remove it completely now.

Before my patch, the mark array contained:

1. Coverage info
2. Flip-flop counts
3. Original iseq
4. Result of once instruction
5. Literals
6. Child iseqs
7. Jump table entries

This patch removes 5, 6, and 7 from the array. Literals and child iseqs are marked by walking the instructions, and I added a loop in the mark function to iterate and mark jump tables.

> I figured this might be the case. This patch should make ISeq marking slower, but I wasn't sure if it would be mitigated by the fact that the ISeq objects get old. It seems like RGenGC solves the problem.

As you two said, it is not problem I also think (hopefully). But I want to know "how slow".

I've attached 2 patches. One is an update so that make test-all passes, second one is a patch test test iterating over the instructions and time iteration. I've also attached a benchmark I used to time instruction iteration. Here is the output:

```
Warming up --------------------------------------
             107 fast    51.019k i/100ms
             987 fast     6.558k i/100ms
            9987 fast   681.000  i/100ms
           99987 fast    68.000  i/100ms
              98 slow    35.413k i/100ms
             998 slow     3.895k i/100ms
           10004 slow   397.000  i/100ms
           99998 slow    39.000  i/100ms
Calculating -------------------------------------
             107 fast    592.822k (± 4.3%) i/s -      2.959M in   5.002552s
             987 fast     67.417k (± 3.5%) i/s -    341.016k in   5.064936s
            9987 fast      6.671k (± 5.4%) i/s -     33.369k in   5.020232s
           99987 fast    653.789  (± 8.3%) i/s -      3.264k in   5.032978s
              98 slow    378.918k (±10.4%) i/s -      1.912M in   5.113554s
             998 slow     39.021k (± 5.3%) i/s -    194.750k in   5.007117s
           10004 slow      3.894k (± 5.3%) i/s -     19.850k in   5.112539s
           99998 slow    387.217  (± 6.7%) i/s -      1.950k in   5.065411s
```

I made a fast test and a slow test. The first number on each line is the size of iseq_encoded. For example, "107 fast" means iseq_encoded is 107 long. I expected the time to be linear with respect to the size of iseq_encoded, so it should get slower in relation to the length of iseq_encoded. The iterator has to translate the encoded iseq back to the original iseq value so that it can get the operand count. Translating the iseq is also a linear search:

https://github.com/ruby/ruby/blob/36d91068ed9297cb792735f93f31d0bf186afeec/iseq.c#L117-L129

So I made a "best case" test, which is the fast test, and a "worst case" test, which is the slow test. The "best case" test contains many getlocal instructions, because they are close to the start in insns.inc. The "worst case" test contains lots of putobject_INT2FIX_1_ instructions, which is near the end of the list.

For the fast test, the mean is 65,491,636 VALUE per second:

```
> mark_perf_fast$pointers * mark_perf_fast$ips
[1] 63431968 66540097 66624108 65370370
> mean(mark_perf_fast$pointers * mark_perf_fast$ips)
[1] 65491636
```

For the slow test, the mean is 38,439,253 VALUE per second:

```
> mark_perf_slow$pointers * mark_perf_slow$ips
[1] 37133984 38943026 38959108 38720894
> mean(mark_perf_slow$pointers * mark_perf_slow$ips)
[1] 38439253
```

Of course, this bench mark doesn't test instructions that have more operands, and iseq objects that don't have anything to mark will still have to scan iseq_encoded to find references.

As I said, I didn't see any performance changes with make gcbench-rdoc. If this time is too slow, I think we can add some optimizations (like a bitmap, or optimization for iseq that have 0 references in iseq_encoded). If there is some other benchmark you have in mind, please let me know!

#5 - 01/23/2018 01:52 AM - tenderlovemaking (Aaron Patterson)

I've been doing more investigation about performance of this patch. Using the same Rails application I used for the initial graphs, I walked the heap for all instruction sequences and output the size of iseq_encoded along with the number of markable objects it found in the encoded iseq. The code I used to walk the heap is here:

https://gist.github.com/tenderlove/63ae5ec669a1e797b611aeaa0b72073e

The data I gathered is here:

https://gist.github.com/tenderlove/d7ea6bb52004f8f5bf10af5b0fcfea4c

Using the numbers from above, I estimate it would take between 15ms and 25ms to mark all ISeq in the Rails application:

```
> sum(iseqs$size) / 65491636
[1] 0.01509967
> sum(iseqs$size) / 38439253
[1] 0.02572636
```

I made a histogram of Instruction Sequences bucketed by the number of markable objects that are in the iseq:

35252871-58bc9700-ff97-11e7-90f3-ec6bc3d8710e.png
Approximately 60% of the instruction sequences have 0 markable objects. Those 60% account for 35% of the total iseq_encoded that needs to be walked:

```
> no_markables <- subset(iseqs, markables == 0)
> length(no_markables$markables) / length(iseqs$markables)
[1] 0.5950202
> sum(no_markables$size) / sum(iseqs$size)
[1] 0.3508831
```

If we set a flag on the iseq at compile time that the iseq "has objects to mark", we could reduce mark time to between 10ms and 16ms for all iseq objects:

```
> has_markables <- subset(iseqs, markables > 0)
> sum(has_markables$size) / 65491636
[1] 0.00980145
> sum(has_markables$size) / 38439253
[1] 0.01669941
```

What do you think?

#6 - 01/28/2018 01:43 PM - ko1 (Koichi Sasada)

> Approximately 60% of the instruction sequences have 0 markable objects. Those 60% account for 35% of the total iseq_encoded that needs to be walked:

Interesting.

Just an idea. Now we allocate Array with extra area and the size of extra area called "capa" (RArray::as::heap::aux::capa). I tried to show the size and capa with the following patch:

```
Index: iseq.c
===================================================================
--- iseq.c  (□□□□□ 62078)
+++ iseq.c  (□□□□□)
@@ -418,6 +418,10 @@
     if (ruby_vm_event_enabled_flags & ISEQ_TRACE_EVENTS) {
     rb_iseq_trace_set(iseq, ruby_vm_event_enabled_flags & ISEQ_TRACE_EVENTS);
     }
+
+    fprintf(stderr, "size:%d, capa:%d\n",
+            (int)RARRAY_LEN(iseq->body->mark_ary),
+            (int)(FL_TEST(iseq->body->mark_ary, RARRAY_EMBED_FLAG) ? 0 : RARRAY(iseq->body->mark_ary)->as.hea
p.aux.capa));
     return Qtrue;
 }
```

and test-all shows:

```
size:3, capa:0
size:11, capa:20
size:4, capa:20
size:4, capa:20
size:6, capa:20
size:3, capa:0
size:5, capa:20
size:4, capa:20
size:5, capa:20
size:8, capa:20
size:4, capa:20
size:5, capa:20
size:8, capa:20
size:3, capa:0
size:5, capa:20
size:3, capa:0
size:5, capa:20
size:6, capa:20
size:3, capa:0
size:5, capa:20
size:6, capa:20
```

```
size:5, capa:20
size:7, capa:20
size:5, capa:20
size:7, capa:20
size:7, capa:20
size:5, capa:20
size:3, capa:0
size:5, capa:20
size:5, capa:20
size:23, capa:37
size:3, capa:0
size:6, capa:20
size:3, capa:0
size:5, capa:20
size:4, capa:20
size:4, capa:20
size:4, capa:20
size:4, capa:20
...
```

(sorry, no formal statistics data)

It seems many extra data we are holding. If we shrink such extra space (capa) just after compiling, it can reduce memory.

Of course, Aaron's approach helps more (because we don't need to keep mark_ary any more for referencing objects). But it is simple and no computation cost on marking phase.

Thoughts?

#7 - 01/28/2018 07:51 PM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> It seems many extra data we are holding. If we shrink such
> extra space (capa) just after compiling, it can reduce memory.

Yes, I think we can resize capa anyways in other places,
perhaps rb_ary_freeze, since we already resize in rb_str_freeze.

Currently we do not freeze mark_ary because of the `once' insn,
but we can mark is->once.value directly, too.

> Of course, Aaron's approach helps more (because we don't need
> to keep mark_ary any more for referencing objects). But it
> is simple and no computation cost on marking phase.

Right, since we have RGenGC to reduce marking; I prefer Aaron's
approach for this.

#8 - 02/06/2018 12:36 AM - tenderlovemaking (Aaron Patterson)

- File direct_marking.diff added

ko1 (Koichi Sasada) wrote:

> Approximately 60% of the instruction sequences have 0 markable objects. Those 60% account for 35% of the total iseq_encoded that needs to be walked:
>
> It seems many extra data we are holding. If we shrink such extra space (capa) just after compiling, it can reduce memory.

I tried this approach to compare, here is the patch I used:

https://gist.githubusercontent.com/tenderlove/f7fe6e09d6f9cacd3299abdd39fb65b7/raw/9b7dd3718278f7be0eb18bedf94590573e313ab2/resize_array.diff

This results in slightly more memory usage than directly marking iseq in my test (57300896 bytes vs 57112281 bytes). Unfortunately this patch also made more calls to malloc / free in order to allocate the smaller array. Maybe there is a way to do this in-place?

> Of course, Aaron's approach helps more (because we don't need to keep mark_ary any more for referencing objects). But it is simple and no computation cost on marking phase.

Thoughts?

I wanted to see what it takes to remove mark_ary and what the benefits would be, so I did that too. I found removing the mark array resulted in a 6.6% memory reduction compared to trunk (vs 3% from direct marking). To remove the mark array, first I started marking once results:

https://github.com/github/ruby/commit/b85d83613ad37d8c56277a7ac7d9bda69e1d8f67.diff

The once instruction declares its operand is an inline cache struct, but it's actually a iseq_inline_storage_entry, so I had to add a conditional to cast the IC but only for once and trace_once instructions. Next I was able to remove the mark array:

https://github.com/github/ruby/commit/29c95eec9c5ce6a9735169c5d6f1d08343d643d1.diff

I added a flag to the iseq to indicate whether or not the iseq has markable objects:

https://github.com/github/ruby/commit/fcb70543fcf73216e01634af1cc6e161eec3d2b6.diff

After that patch the only iseqs that have performance impact are ones that actually need to be disassembled for marking. Finally, I added a new operand for once instructions, then removed the conditionals from the mark / compile function:

https://github.com/github/ruby/commit/78ce485d9880e8694d586e93a48429f4aa907c18.diff

I ran the same benchmark to compare all four versions:

1. trunk
2. Directly marking
3. Removing the mark array
4. Resizing the mark array

35835276-9d8b37cc-0a8e-11e8-8a46-5ba391c0e666.png
From the graph you can see that resizing the mark array ends up with the same memory usage as direct marking, but takes more malloc calls to do it (the X axis is process size at each malloc, so the more samples the more malloc calls).

Out of curiosity, I printed GC.stat after booting the Rails app for trunk, resizing the array, and removal:

key	trunk	resize	remove
count	46	47	49
heap_allocated_pages	664	656	505
heap_sorted_length	664	656	505
heap_allocatable_pages	0	0	0
heap_available_slots	270637	267387	205847
heap_live_slots	184277	184275	157995
heap_free_slots	86360	83112	47852
heap_final_slots	0	0	0
heap_marked_slots	184194	184192	157912
heap_eden_pages	664	656	505
heap_tomb_pages	0	0	0
total_allocated_pages	664	656	505
total_freed_pages	0	0	0
total_allocated_objects	986460	1005514	951828
total_freed_objects	802183	821239	793833
malloc_increase_bytes	1168	1168	1168
malloc_increase_bytes_limit	16777216	16777216	16777216
minor_gc_count	36	36	39
major_gc_count	10	11	10
remembered_wb_unprotected_objects	1640	1639	1660
remembered_wb_unprotected_objects_limit	3280	3278	3320
old_objects	177699	177482	153681
old_objects_limit	355398	354964	307362
oldmalloc_increase_bytes	1616	1616	2522576
oldmalloc_increase_bytes_limit	16777216	16777216	16777216

As expected, removing the mark array decreased the number of allocated pages and the number of live slots.

Anyway, I really like the idea of removing the mark array, though I realize this patch is a little complex. I think the savings are worth the change. I've attached an updated diff with all of the changes I've made so far.

As a side note, I think we could remove original_iseq from the struct / mark array. AFAICT the original_iseq slot is just there because rb_iseq_original_iseq returns the underlying pointer and something needs to keep it alive. Maybe we could use a callback and keep the reference on the stack? I will send a patch for it so we can discuss on a different ticket. :)

#9 - 02/06/2018 02:21 AM - normalperson (Eric Wong)

tenderlove@ruby-lang.org wrote:

> https://gist.githubusercontent.com/tenderlove/f7fe6e09d6f9cacd3299abdd39fb65b7/raw/9b7dd3718278f7be0eb18bedf94590573e313ab2/resize_array.diff
>
> Unfortunately this patch also made more calls to malloc / free
> in order to allocate the smaller array. Maybe there is a way
> to do this in-place?

Not really. At least I do not recall malloc implementations
being optimized for shrinking smaller allocations with realloc.
For multi-page allocations, there's mremap(2), so it's not
appropriate for most Ruby arrays (and also Linux-specific).

Anyways, I like the rest of your changes to remove mark_ary
and flagging markability.

#10 - 02/21/2018 06:11 AM - ko1 (Koichi Sasada)

I realize my misunderstanding on aaron's proposal and now I'm very positive on it.
Thank you aaron!

(my misunderstanding: prepare all mark functions for each instructions. it should be hard to maintain)

#11 - 03/05/2018 08:25 PM - tenderlovemaking (Aaron Patterson)

ko1 (Koichi Sasada) wrote:

> I realize my misunderstanding on aaron's proposal and now I'm very positive on it.
> Thank you aaron!
>
> (my misunderstanding: prepare all mark functions for each instructions. it should be hard to maintain)

Cool. Do you mind if I apply the patch? (Just checking □□)

#12 - 03/11/2018 12:47 AM - tenderlovemaking (Aaron Patterson)

- Status changed from Open to Closed

Applied in r62706

#13 - 03/12/2018 06:37 AM - nobu (Nobuyoshi Nakada)

- Related to Bug #14596: Ruby master is broken with bootsnap added

Files

iseq_mark.diff	6.28 KB	01/17/2018	tenderlovemaking (Aaron Patterson)
iseq_mark.diff	6.28 KB	01/18/2018	tenderlovemaking (Aaron Patterson)
iseq_mark.diff	7.26 KB	01/20/2018	tenderlovemaking (Aaron Patterson)
benchmark_methods.diff	1.23 KB	01/20/2018	tenderlovemaking (Aaron Patterson)
bench.rb	3.01 KB	01/20/2018	tenderlovemaking (Aaron Patterson)
direct_marking.diff	18.4 KB	02/06/2018	tenderlovemaking (Aaron Patterson)