Ruby - Feature #8259

Atomic attributes accessors

04/12/2013 05:43 PM - funny_falcon (Yura Sokolov)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Motivated by this gist ((URL:https://gist.github.com/jstorimer/5298581)) and atomic gem

I propose Class.attr_atomic which will add methods for atomic swap and CAS:

```
class MyNode
  attr_accessor :item
  attr_atomic :successor

  def initialize(item, successor)
    @item = item
    @successor = successor
  end
end
node = MyNode.new(i, other_node)

# attr_atomic ensures at least #{attr} reader method exists. May be, it should
# be sure it does volatile access.
node.successor

# #{attr}_cas(old_value, new_value) do CAS: atomic compare and swap
if node.successor_cas(other_node, new_node)
  print "there were no interleaving with other threads"
end

# #{attr}_swap atomically swaps value and returns old value.
# It ensures that no other thread interleaves getting old value and setting
# new one by cas (or other primitive if exists, like in Java 8)
node.successor_swap(new_node)
```

It will be very simple for MRI cause of GIL, and it will use atomic primitives for
other implementations.

Note: both (({#{attr}_swap})) and (({#{attr}_cas})) should raise an error if instance variable were not explicitly set before.

Example for nonblocking queue: ((URL:https://gist.github.com/funny-falcon/5370416))

Something similar should be proposed for Structs. May be override same method as (({Struct.attr_atomic}))

Open question for reader:
should (({attr_atomic :my_attr})) ensure that #my_attr reader method exists?
Should it guarantee that (({#my_attr})) provides 'volatile' access?
May be, (({attr_reader :my_attr})) already ought to provide 'volatile' semantic?
May be, semantic of (({@my_attr})) should have volatile semantic (i doubt for that)?

Related issues:

Related to Ruby - Feature #12019: Better low-level support for writing concur...	**Assigned**

History

#1 - 04/13/2013 01:27 AM - headius (Charles Nutter)

Great to see this proposed officially!

I implemented something very much like this for JRuby as a proof-of-concept. It was in response to thedarkone's recent work on making Rails truly thread-safe/thread-aware.

My feature was almost exactly like yours, with an explicit call to declare an attribute as "volatile" (the characteristic that makes atomic operations possible). Doing so created a _cas method, made accessors do volatile operations, and I may also have had a simple atomic swap (getAndSet). CAS is probably enough to add, though.

thedarkone had concerns about my proposed API. I believe he wanted to be able to treat *any* variable access as volatile (even direct access via @foo = 1) or perhaps he simply didn't like having to go through a special method. I'll try to get him to comment here.

One concern about CAS (which has actually become an issue for my "atomic" gem): treatment of numerics. Specifically, what does it mean to CAS a numeric value when numeric idempotence varies across implementations:

MRI 1.9.x and lower only have idempotent signed fixnums up to 31 bits on 32-bit builds and 63 bits on 64-bit builds. Rubinius and MacRuby follow suit.

MRI 2.0.0 has idempotent floats only on 64-bit and only up to some number of bits of precision (is that correct?). MacRuby does something similar.

I believe MagLev has fixnums but not flonums. Unsure.

JRuby has idempotent fixnums only up to 8 bits (signed) due to the cost of caching Fixnum objects (JVM does not have fixnums, so we have to mitigate the cost of objects).

Topaz does not have fixnums or flonums and relies on escape analysis/detection to eliminate Fixnum objects.

IronRuby does something similar to JRuby, but could potentially make Fixnums and Floats be value types; I'm not sure if this would make them idempotent or not.

And this all ignores the fact that Fixnum transparently overflows into Bignum, which is represented as a full, non-idempotent object on all implementations.

So we've got a case where this code would start to fail at different times on different implementations:

```
number = obj.number
success = obj.number_cas(number, number + 1)
fail unless success
```

In the atomic gem, I'm going to be adding AtomicInteger and AtomicFloat for this purpose that either use value equality rather than reference equality (at potentially greater cost) or limit the value range of integers to 64 bits.

Other concerns:

- The JVM does not, until Java 8, have a way to insert an explicit memory barrier into Java code without having a volatile field access or a lock acquisition (which does volatile-like things). Even in Java 8, it is via a non-standard "fences" API. JRuby currently uses it to improve volatility guarantees of instance variables. On Java 6 and 7 we fall back on a slower implementation that uses explicit volatile operations on a larger scale.
- The JVM also does not provide a way to make only a single element of an array be volatile, but you can use nonstandard back-door APIs to simulate it (which is what AtomicReferenceArray and friends do).
- JVM folks have introduced the concept of a "lazy set" which is intended to mean you don't really expect full volatile semantics for this write (and don't want to pay for volatile semantics every time).
- Optimizing implementations may get to a point where they can optimize away repeated accesses of instance variables. In the Java world, these optimizations are limited by the volatile field modifier and the Java Memory Model, which inserts explicit ordering and visibility constraints on volatile accesses. It would seem to me that Ruby needs to more formally define volatile semantics along with adding this feature.

That's all I have for now :-)

#2 - 04/13/2013 11:55 PM - funny_falcon (Yura Sokolov)

I think, @ivar access should not be volatile as in any other language,
but obj.ivar could be volatile if attr_atomic :ivar were called.

Number idempotention should not be a great problem cause most of time the same
old object is used for CAS. But, yeah, we could treat numbers as a special case,
and do two step CAS (ruby-like pseudocode):

```
def ivar_cas(old, new)
  if Number === old
    stored = @ivar
    if stored == old
      ivar_hardware_cas(stored, new)
    end
  else
    ivar_hardware_cas(old, new)
  end
```

```
end
```

But I could not help with JVM internals :(

#3 - 04/16/2013 06:34 AM - headius (Charles Nutter)

funny_falcon (Yura Sokolov) wrote:

> I think, @ivar access should not be volatile as in any other language,
> but obj.ivar could be volatile if attr_atomic :ivar were called.

Agreed. The dynamic nature by which @ivar can be instantiated makes marking them as volatile very tricky, on any implementation.

> Number idempotention should not be a great problem cause most of time the same
> old object is used for CAS. But, yeah, we could treat numbers as a special case,
> and do two step CAS (ruby-like pseudocode):

```
def ivar_cas(old, new)
  if Number === old
    stored = @ivar
    if stored == old
      ivar_hardware_cas(stored, new)
    end
  else
    ivar_hardware_cas(old, new)
  end
end
```

This logic would be sufficient in JRuby as well, but comes with a fairly high cost: an === call even when the value is non-numeric.

The same logic implemented natively in the atomic accessors would probably be simple enough to optimize (e.g. in JRuby it would be an instanceof RubyNumeric check).

#4 - 04/16/2013 08:09 AM - headius (Charles Nutter)

FYI, link to a current issue with the atomic gem I'm fixing using a loop + == + CAS: https://github.com/headius/ruby-atomic/issues/19

#5 - 04/16/2013 10:50 AM - nobu (Nobuyoshi Nakada)

- Description updated

#6 - 04/16/2013 10:59 AM - nobu (Nobuyoshi Nakada)

Why do you consider comparison atomic?

#7 - 04/16/2013 01:42 PM - funny_falcon (Yura Sokolov)

Comparison is not atomic. It is used to be ensure, we could use value, stored in @ivar for real CAS. Semantic of method at whole doesn't change, cause if comparison fails, then CAS will fail also.

#8 - 04/16/2013 01:54 PM - headius (Charles Nutter)

Comparison of two numeric values *should* be consistent and unchanging, or else I feel that various contracts of numbers are being violated. In Java, this is handled by having numeric values be primitives, and therefore all representations of equality are consistent. In Ruby, where some numerics are idempotent and some are not, I think it is reasonable to extend the CAS operation to do a value equality check. So, the contract would be:

- For non-numeric types, CAS checks only reference equality (hardware CAS).
- For numeric types, CAS checks value equality (using reference equality -- hardware CAS -- to ensure nothing has changed while checking value equality).

This is how version 1.1.8 of the atomic gem will work, once I (or someone else) implements value equality CAS for the C ext.

#9 - 04/16/2013 11:54 PM - headius (Charles Nutter)

I have completed adding the numeric logic to the atomic gem and pushed 1.1.8.

The version for JRuby is here: https://github.com/headius/ruby-atomic/blob/master/ext/org/jruby/ext/atomic/AtomicReferenceLibrary.java#L129

The version for MRI, Rubinius, and others is here: https://github.com/headius/ruby-atomic/blob/master/lib/atomic/numeric_cas_wrapper.rb

#10 - 04/17/2013 03:53 AM - funny_falcon (Yura Sokolov)

Charles, I really sure there is no need for while true in your numeric

handling cas -
the nature of cas is "change if no one changes yet", so that your while true violates natures of cas.

2013/4/16 headius (Charles Nutter) headius@headius.com

> Issue #8259 has been updated by headius (Charles Nutter).
>
> I have completed adding the numeric logic to the atomic gem and pushed
> 1.1.8.
>
> The version for JRuby is here:
> https://github.com/headius/ruby-atomic/blob/master/ext/org/jruby/ext/atomic/AtomicReferenceLibrary.java#L129
>
> ## The version for MRI, Rubinius, and others is here:
> https://github.com/headius/ruby-atomic/blob/master/lib/atomic/numeric_cas_wrapper.rb
>
> Feature #8259: Atomic attributes accessors
> https://bugs.ruby-lang.org/issues/8259#change-38617
>
> Author: funny_falcon (Yura Sokolov)
> Status: Open
> Priority: Normal
> Assignee:
> Category:
> Target version:
>
> =begin
> Motivated by this gist ((URL:https://gist.github.com/jstorimer/5298581))
> and atomic gem
>
> I propose Class.attr_atomic which will add methods for atomic swap and CAS:
>
> class MyNode
> attr_accessor :item
> attr_atomic :successor
>
> ```
> def initialize(item, successor)
>   @item = item
>   @successor = successor
> end
> ```
>
> end
> node = MyNode.new(i, other_node)
>
> ## attr_atomic ensures at least #{attr} reader method exists. May be, it
>
> should
>
> ## be sure it does volatile access.
>
> node.successor
>
> ## #{attr}_cas(old_value, new_value) do CAS: atomic compare and swap
>
> if node.successor_cas(other_node, new_node)
> print "there were no interleaving with other threads"
> end
>
> ## #{attr}_swap atomically swaps value and returns old value.
>
> ## It ensures that no other thread interleaves getting old value and
>
> setting
>
> ## new one by cas (or other primitive if exists, like in Java 8)
>
> node.successor_swap(new_node)
>
> It will be very simple for MRI cause of GIL, and it will use atomic

primitives for
other implementations.

Note: both (({#{attr}_swap})) and (({#{attr}_cas})) should raise an error
if instance variable were not explicitly set before.

Example for nonblocking queue: ((<URL:
https://gist.github.com/funny-falcon/5370416>))

Something similar should be proposed for Structs. May be override same
method as (({Struct.attr_atomic}))

Open question for reader:
should (({attr_atomic :my_attr})) ensure that #my_attr reader method
exists?
Should it guarantee that (({#my_attr})) provides 'volatile' access?
May be, (({attr_reader :my_attr})) already ought to provide 'volatile'
semantic?
May be, semantic of (({@my_attr})) should have volatile semantic (i doubt
for that)?
=end

--
http://bugs.ruby-lang.org/

#11 - 04/17/2013 03:57 AM - headius (Charles Nutter)

The "while true" loop is there in order to re-check if the value is == after a change. My justification is that the only atomic part of this is the final CAS, but we want to pretend that the whole == + CAS is atomic; so this loops until either the current value is non-numeric, non-equal, or numeric + equal + has not changed since we last got it.

This pattern is used fairly often in the concurrency utilities on JVM for performing non-atomic logic surrounding an atomic update. Without the loop, an update that happens after the == check and before the CAS but which does not change the *value* of the currently-referenced object would fail. I don't think it should.

#12 - 04/18/2013 01:00 AM - dbussink (Dirkjan Bussink)

What I'm wondering is, do we want to enforce the overhead of numeric CAS for all applications of CAS? Also in the case of numeric handling, the pattern in which I've used CAS most often is that I base the old value on the existing one, which of course still works fine for CAS operations on references.

What I see from this discussion is perhaps two API's. One that is basically identity based and one that is equality based. Wouldn't it be a better idea to provide these two api's separate? That case we don't have to special case numeric handling and people also get equality like handling for non-Numeric classes which would work like the numeric logic here. People can then decide which kind of CAS they need based which kind if comparison they need.

#13 - 04/18/2013 01:08 AM - headius (Charles Nutter)

dbussink (Dirkjan Bussink) wrote:

> What I'm wondering is, do we want to enforce the overhead of numeric CAS for all applications of CAS? Also in the case of numeric handling, the pattern in which I've used CAS most often is that I base the old value on the existing one, which of course still works fine for CAS operations on references.

To be clear, in the Ruby impls of numeric CAS, the only additional cost for non-numerics is a kind_of? check.

In JRuby, it's an instanceof check, which is pretty darn fast.

> What I see from this discussion is perhaps two API's. One that is basically identity based and one that is equality based. Wouldn't it be a better idea to provide these two api's separate? That case we don't have to special case numeric handling and people also get equality like handling for non-Numeric classes which would work like the numeric logic here. People can then decide which kind of CAS they need based which kind if comparison they need.

That would certainly avoid overhead in the non-numeric case, but I worry it would lead to too much confusion. People would forget about the equality CAS and use the other one and get weird bugs because they didn't have the same numeric object in hand. It's also hard to track whether you actually have the same object, since most impls emulate the same value / same object_id behavior from MRI.

In the Atomic gem, I still think it's valid to explicitly have AtomicInteger and AtomicFloat to speed up how those are handled (we can use native CAS against 64-bit long and double rather than against the object reference), but this is a case where it seems like everyone would expect numbers to CAS based on equality rather than reference identity, and not doing it will lead to neverending complaints. I could be wrong.

#14 - 04/18/2013 02:26 AM - dbussink (Dirkjan Bussink)

I highly doubt the neverending complaints case, since this I think people using CAS would usually know what they are doing (at least my experience with using constructs like this). The overhead is actually bigger for the numeric case where a CAS would work for example for Fixnum on MRI and Rubinius without the extra checks. That could of course be optimized in implementations for those platforms.

If you're worried about confusion between equality and identity, we could also have an equality based CAS be the default and have the possibility of using an identity based version if people know that is what they want.

#15 - 04/18/2013 02:50 AM - headius (Charles Nutter)

dbussink (Dirkjan Bussink) wrote:

> I highly doubt the neverending complaints case, since this I think people using CAS would usually know what they are doing (at least my experience with using constructs like this). The overhead is actually bigger for the numeric case where a CAS would work for example for Fixnum on MRI and Rubinius without the extra checks. That could of course be optimized in implementations for those platforms.

You may be right about complaints...I use atomics all the time but I'm unusual. Skewed viewpoint, perhaps.

I'm not sure what you mean by "the overhead is actually bigger". It's a kind_of? type check at worst...is that expensive in Rubinius?

Also, Fixnum will *not* work consistently on MRI or Rubinius without extra checks if any of the values are close to the Fixnum boundary. Optimization specific to those impls would still have to confirm the Fixnum is within a certain range. Perhaps working with Fixnums that are at the failover point into Bignum is not common, but you can't just omit those checks. And you have to know you're dealing with a Fixnum anyway...so you have to check every time.

My justification for doing it unconditionally for all numerics is largely because of the overflow into Bignum. Ruby pretends that integers are one continuum, but only part of that continuum (varying across impls and architectures) is actually idempotent. As a result, all integers would need some portion of the equality checking logic on every implementation.

> If you're worried about confusion between equality and identity, we could also have an equality based CAS be the default and have the possibility of using an identity based version if people know that is what they want.

That's not a bad option, I guess. The main problem here is that I feel like people expect numerics of equal value to essentially be identical, and that's not the case for most numerics on most implementations. If people think they're essentially identical, they might expect CAS to work properly. I don't believe people would have that expectation of non-numerics, so extending equality CAS to all types seems like overkill.

#16 - 04/18/2013 03:06 AM - headius (Charles Nutter)

Having some discussions with dbussink on IRC...

I think the most universally correct option is to have two different paths for reference CAS and value CAS, and rely upon users to choose the right one. As dbussink points out, the people who are going to use atomic ivars are much more likely to know what they're doing.

We don't want to have an additional method for ever single atomic ivar, so perhaps a parameter?

class MyClass
atomic_accessor :foo

```
# generates CAS accessor similar to...
def foo_cas(old, new, compare_value = false)
  ...
end
```

end

mc = MyClass.new
mc.foo = 5 # fixnum, so we'll want value CAS
...

true causes value comparison CAS like I implemented in atomic gem

false does reference CAS as normal, with variable behavior for some numeric values.

mc.foo_cas(5, 6, true)

I can't imagine two separate methods getting approved but this form seems like it would be acceptable.

#17 - 10/02/2013 06:14 AM - headius (Charles Nutter)

- Target version set to Ruby 2.1.0

Trying to wake this one up in hopes of getting it into 2.1. Is there any chance?

Forgive me if I'm breaking process somehow, but ko1 told me to mark the issues I want in 2.1 with Target version=2.1, so I've been doing that.

#18 - 06/09/2014 07:42 AM - headius (Charles Nutter)

Waking this up again. Interest in gems like "atomic" and "concurrent-ruby" has continued to grow rapidly, and Ruby needs to start provided low-level concurrency constructs users need to build high-level constructs. That means atomics, tuples, efficient locks, read-write-locks, and more all need VM-level help.

Let's get this discussion going and ship something in 2.2.

#19 - 06/09/2014 10:05 AM - headius (Charles Nutter)

I have done a prototype of atomic variable accessors in JRuby.

Here's the IRB session. I opted to go with an :atomic option passed to attr_accessor rather than a separate method, but I don't have a strong preference. Atomic operations don't mean much in the context of read-only or write-only, so attr_atomic may be more appropriate.

https://gist.github.com/headius/252206b478018d71d85b

The patch is local to my working copy of JRuby right now, and it needs some work and testing, but this seems like a reasonable way to move forward. I'd really like to see this get into 2.2, so we can start building Ruby concurrency utilities with VM-level support for atomic instance variable operations.

#20 - 06/09/2014 10:09 AM - headius (Charles Nutter)

I should also point out that my implementation is using referential equality. I now think the default should be referential equality, perhaps (after some discussion and debate) with an option to do value equality. Of course, the value-equality cas and swap can be implemented in terms of the referential-equality versions.

#21 - 06/09/2014 11:12 AM - normalperson (Eric Wong)

I'm not sure if setting the attribute on the ivar is a good way to go.
Entries in structs, arrays, hashes, etc may also benefit from atomic
operations (or at least I would like that).

Maybe something like:

```
old = hash["foo"] x= new # swap
hash["foo"] ?x= old : new # cas
```

#22 - 06/09/2014 06:22 PM - normalperson (Eric Wong)

Joel VanderWerf joelvanderwerf@gmail.com wrote:

> On 06/09/2014 04:06 AM, Eric Wong wrote:
>
> > I'm not sure if setting the attribute on the ivar is a good way to go.
> > Entries in structs, arrays, hashes, etc may also benefit from atomic
> > operations (or at least I would like that).
> >
> > Maybe something like:
> >
> > old = hash["foo"] x= new # swap
> > hash["foo"] ?x= old : new # cas
>
> Do you mean that x stands for a symbol to be chosen later?

No, x being short for xchg. And ?x= being cmpxchg
(and trying to look like a ternary operation)

#23 - 06/09/2014 10:23 PM - thedarkone (Vit Z)

In my opinion an ideal API would have "atomic" attributes declared at the class level,
yet can be used as normal @foo i-vars:

```
class MyNode
  atomic :item, :successor

  def initialize(item, successor)
    @foo = :foo # this is plain old i-var
```

```
    @item = item # this is atomic
    @successor = successor # this is atomic
    puts @foo # this is a non-atomic read
    puts @item # this is an atomic read
  end

  # this creates reader and writer that are atomic, because :item is atomic
  attr_accessor :item
  # this is not atomic
  attr_accessor :foo

  def cas_new_item(expected_old_item, new_item)
    # atomic :item also generates a new private cas_item method
    # that implements compare-and-swap semantics
    cas_item(expected_old_item, new_item)
  end

  class << self
    # atomic i-vars can also be declared on Modules/Classes
    atomic :head
  end

  @head = nil
  # cas_head is also added as a private module method
  cas_head(nil, MyNode.new(:item, nil))
end
```

Another thing, and I know that this might be controversial, is that atomic attributes should only be "declare-able" before a first instance of the class is created:

```
class OtherNode
  new
  # this should raise an ArgumentError, since an instance of OtherNode
  # has already been created
  atomic :item

  class << self
    # not a problem on Module/Class level
    atomic :head
  end
end
```

I believe this is absolutely necessary in order to have a proper and performant implementation of "atomic" on concurrent Ruby VMs. If I turn out to be wrong the restriction can be always be removed, however if it is not in place and I am right, this forever puts an upper-limit constraint on Ruby performance in concurrent environments.

The auto-generated cas_item methods need to be "dumb" pointer swapping CASes, ie no clever trickery checking wether passed-in arguments are Fixnums or something else, this is again necessary to not hamstring future upper limit on performance.

The other thing that is needed is an "atomic" fixed-size array, lets call it AtomicTuple. It has to be fixed-size again because otherwise this would create problems for concurrent Ruby VMs.

```
tuple = AtomicTuple.new(16) # creates a new 16-slotted atomic array
# an atomic read
tuple[0] # => nil
# an atomic write
tuple[0] = :foo
# a cas
tuple.cas(0, :foo, :bar) # => true
tuple[0] # => :bar
```

I'm fairly certain that adding this to MRI should be quite easy, because of the GVL all the "atomicness" is already present. For example declaring an i-var atomic (via atomic :item) doesn't have to do anything except to add a cas_item method, after all i-vars are already atomic. cas_item also in the first implementation doesn't have to use native cmpxchg assembly a simple pointer check-then-swap should be fine (as long as this is in C GVL has our back).

Eric Wong:
Adding "atomic" operations to built in Arrays, Hashes etc. is a very bad idea because that would force that *all* of the methods on Arrays and Hashes be

concurrency friendly and would result in irreparable performance problems for truly concurrent Ruby VMs (and this again would put an upper limit on Ruby performance).

-thedarkone

#24 - 06/09/2014 11:43 PM - normalperson (Eric Wong)

thedarkone2@gmail.com wrote:

> Adding "atomic" operations to built in Arrays, Hashes etc. is a very bad idea because that would force that *all* of the methods on Arrays and Hashes be concurrency friendly and would result in irreparable performance problems for truly concurrent Ruby VMs (and this again would put an upper limit on Ruby performance).

I do atomic operations all the time in C on arbitrary addresses.
Lazy, non-atomic accesses run without speed penalty if I don't need
up-to-date data.

The uncommon case of Array/Hash shrinkage would require RCU or similar
(epoch-based reclamation). But there's no penalty for reads or in-place
modifications other than the cost of the atomic and required memory
barriers.

Ruby swap/cas (in Ruby) should probably raise on non-existent Array/Hash
elements.

#25 - 06/10/2014 09:39 AM - thedarkone (Vit Z)

> I do atomic operations all the time in C on arbitrary addresses.
> Lazy, non-atomic accesses run without speed penalty if I don't need
> up-to-date data.

Are you talking about Ruby with GVL or C in general? If C in general then
I don't understand how barrier-less access to concurrently updatable
data does not result in unexpected behaviors for you...

> The uncommon case of Array/Hash shrinkage would require RCU or similar
> (epoch-based reclamation). But there's no penalty for reads or in-place
> modifications other than the cost of the atomic and required memory
> barriers.

The cost of memory barriers is what I was talking about, right now
concurrent Ruby VMs don't need to have any memory barriers on
any of the Array/Hash methods. Adding "atomic" methods
to Array/Hash would force them to put some kind of memory
barriers on all of the methods. This will result in a performance penalty
that cannot be avoided. What I am worried about, is that this will result
in native Ruby Array/Hash becoming slower for single-threaded
usage forever and there will no way to ever get the original performance
back.

-thedarkone

#26 - 06/10/2014 07:21 PM - normalperson (Eric Wong)

thedarkone2@gmail.com wrote:

> Eric Wong wrote:
>
> > I do atomic operations all the time in C on arbitrary addresses.
> > Lazy, non-atomic accesses run without speed penalty if I don't need
> > up-to-date data.
>
> Are you talking about Ruby with GVL or C in general? If C in general then
> I don't understand how barrier-less access to concurrently updatable
> data does not result in unexpected behaviors for you...

"atomic" is a bad word to describe the features we want here.
We really want "synchronized" access, not "atomic" access.

C in general. Aligned access on word-sized types (which VALUE is in C Ruby) is atomic in the sense we won't see data which is in the middle of an update. Yes, we may see stale data; but we cannot see a value which is in the middle of an update.

If there are bitfields causing unaligned access or if we need to access 64-bit types on a 32-bit system, we cannot access those atomically without a lock.

> The uncommon case of Array/Hash shrinkage would require RCU or similar (epoch-based reclamation). But there's no penalty for reads or in-place modifications other than the cost of the atomic and required memory barriers.

> The cost of memory barriers is what I was talking about, right now concurrent Ruby VMs don't need to have any memory barriers on any of the Array/Hash methods. Adding "atomic" methods to Array/Hash would force them to put some kind of memory barriers on all of the methods. This will result in a performance penalty that cannot be avoided. What I am worried about, is that this will result in native Ruby Array/Hash becoming slower for single-threaded usage forever and there will no way to ever get the original performance back.

Right. There's no way I will ever advocate a memory barrier of any kind by default for reads or in-place updates.

Unfortunately, changing capacity of an array or hash is tricky and probably requires barriers for most or all cases (unless escape analysis can elide barriers, but that's pie-in-the-sky territory).

#27 - 06/10/2014 09:35 PM - thedarkone (Vit Z)

Eric Wong wrote:

> Right. There's no way I will ever advocate a memory barrier of any kind by default for reads or in-place updates.
>
> Unfortunately, changing capacity of an array or hash is tricky and probably requires barriers for most or all cases (unless escape analysis can elide barriers, but that's pie-in-the-sky territory).

My thesis and objection to adding "atomic" methods to Array or Hash is that it would necessarily entail making them thread-safe as a whole. However making them thread-safe on concurrent Ruby VMs is costly and carries a performance penalty even for a single threaded usage. It will also be impossible to undo (once Ruby declares Hash to be safe to use concurrently, there is no going back on this).

I am all for addition of ConcurrentHash or Concurrent::Array (these new data structures would have cas and swap methods), but for performance reasons plain old Hash and Array should be kept completely un-thread-safe.

#28 - 06/10/2014 10:00 PM - normalperson (Eric Wong)

thedarkone2@gmail.com wrote:

> I am all for addition of ConcurrentHash or Concurrent::Array (these new data structures would have cas and swap methods), but for performance reasons plain old Hash and Array should be kept completely un-thread-safe.

Does that mean segfaulting the VM on concurrent Hash or Array access is OK? I don't think any current Ruby VM allows that.

If we need to prevent segfaults on concurrent access, I suspect we'll

need to pay some concurrency costs.

Personally, I'm OK if we allow the VM to segfault on concurrent accesses, but I doubt others will agree with me.

#29 - 06/11/2014 08:14 AM - thedarkone (Vit Z)

Eric Wong wrote:

> Does that mean segfaulting the VM on concurrent Hash or Array access is OK? I don't think any current Ruby VM allows that.
>
> If we need to prevent segfaults on concurrent access, I suspect we'll need to pay some concurrency costs.
>
> Personally, I'm OK if we allow the VM to segfault on concurrent accesses, but I doubt others will agree with me.

I agree with you :).

Otherwise JRuby is implemented on top of JVM, which is supposed to be a safe language/VM, so one should never be able to segfault JVM and out-of-thin-air values are guaranteed not to happen. However anything else goes (uninitialized objects, exceptions, missed writes, etc. for example Hash not returning stored values, or returning wrong mapping, getting into corrupt state and going into infinite loop or throwing exception on every access, all this is allowed).

As for Rubinius I would think it probably strives to copy JVM guarantees, therefore all of the above applies.

#30 - 06/14/2014 02:44 PM - pitr.ch (Petr Chalupa)

Hello guys,
it certainly looks appealing and convenient to have atomic ivars, but I have an devil advocate's question. What are the advantages of having atomic_attr over just using @a = Atomic.new('a_value') with some convenience methods like def a_swap(v); @a.swap(v); end added? Is it a performance? How much faster is it? To me using Atomic seems perfectly fine, I like clarity without confusion. It does not seem right that reading a @var would behave differently depending on attr_reader atomic: true||false.

#31 - 06/14/2014 03:25 PM - headius (Charles Nutter)

Responding to recent comments...

- Atomic operations for Array, Hash

I lean toward introducing purpose-built collections for this. The current Array and Hash are very fast and about as lightweight as they can be for their features. Making them support atomic operations and thread-safe access would either make them considerably larger or considerably slower. Larger, because for example Array's internal array of elements and size *must* be updatable atomically, and the only way to do that is to have a structure wrap them. That means at least one more indirection to get to array elements, and a new object allocation every time you want to grow the internal array. Doable, but obviously not as lightweight *or* as fast as what we have now. A similar statement goes for Hash...if you need to update the linked buckets (because remember, we have to maintain insertion order too!) or the bucket array (same problem as in Array), you need to be able to do it atomically. And in both cases, you can only avoid the extra indirection by fully locking all reads and writes, which then quickly becomes a bottleneck.

If we want atomic/threadsafe collections, they really need to have separate implementations.

- Why introduce atomic ivars/attrs rather than just using Atomic

Atomic works just fine *functionally*, but for every reference you want to update atomically you need to have an Atomic Object around it. on JRuby, on a 64-bit JVM, this is dozens of bytes of additional allocation. It's also an extra indirection we don't want to have for fast atomic access.

- On reference equality only, rather than value equality

I agree as well...we need to expect that all objects are unique pointers, and that they will be compared by identity. Of course, on MRI two different Float or Fixnum "objects" will be identical if they have the same value, while only Fixnums are identical in Rubinius, and only a small range of Fixnums are identical in JRuby. We could, however, add a form of atomic ivar CAS that also considers value, and people can opt into it when they're dealing with e.g. numbers, strings expected to be the same, etc. Honestly, though, value equality is orthogonal to atomic updates...it would be better for users to use the basic reference CAS and then put their own logic around it to check value. See the CAS implementation in the Atomic gem.

- Whether core collections are threadsafe today

They are not, in any implementation. However, MRI prevents *all* concurrency at the Ruby level, so you never notice that the collections aren't thread-safe. JRuby and Rubinius make no guarantees about Array, Hash, String thread-safety. They may not segfault (JVM at least has guards to prevent dereferencing invalid pointers), but they won't work as expected under concurrent modification.

- Atomic versus synchronized versus volatile versus...

I'm sure I'll end up using JVM Memory Model terminology...since no other Ruby implementation has a formal memory model. The definitions on the JVM, roughly:

volatile: Mostly involves ordering of reads and writes. Writes to a variable marked as volatile are guaranteed to "happen" -- propagate throughout the CPU/caches -- before any reads that happen afterwards. It's a little abstract, but basically it's guaranteeing that when you make an update of a volatile variable, any threads that look at that variable will see the value you just wrote, assuming nobody else is writing.

atomic: Performing a read + write operation, perhaps with a comparison, as a single operation that either succeeds or fails as a whole. This is your compare-and-swap, get-and-set (swap), increment-and-get, and so on.

synchronized: Synchronization guarantees that a given body of code will only be executed by a single thread + object combination. So if you have an array in hand and you synchronize access to it, that array's mutators won't run on more than one thread at once. This is not the same as either volatile or atomic, though it can be implemented atop them. Equivalent concept in MRI would be using a mutex or monitor (JVM basically provides a built-in reentrant monitor + condvar on every object).

#32 - 06/14/2014 07:09 PM - thedarkone (Vit Z)

Petr Chalupa wrote:

> it certainly looks appealing and convenient to have atomic ivars, but I have an devil advocate's question. What are the advantages of having atomic_attr over just using @a = Atomic.new('a_value') with some convenience methods like def a_swap(v); @a.swap(v); end added? Is it a performance?

Performance and memory efficiency, plus some minor issues with pre-initialization (in your example @a = Atomic.new) in constructor and safe-publication issues.

> To me using Atomic seems perfectly fine, I like clarity without confusion. It does not seem right that reading a @var would behave differently depending on attr_reader atomic: true||false.

Note that by "behave differently" you mean "will not break under concurrent access". Besides I would like to point you at Java experience, where volatile (Java's "atomic" i-vars) and normal instance variables also look the same this.foo; vs this.foo;, I'm not aware that this is creating any problems.

On reference equality only, rather than value equality

I would like to echo Dirkjan Bussink's experience from his post above, when it comes to CAS - one usually reads the old value and then tries to CAS in a new value (using the previously read value), in such use cases reference vs value equality is not an issue.

The other typical CAS use case are state machines, where "atomic" values are integers representing a state:

```
class ReadWriteLock
  def initialize
    # @state == 0 means unlocked
    # @state == 1 means read locked
    # @state == 2 means write locked
    @state = 0
  end

  def lock_for_read!
    cas_state(0, 1)
  end

  def lock_for_write!
    cas_state(0, 2)
  end

  def read_locked?
    @state == 1
  end

  def write_locked?
    @state == 2
  end
end
```

In such a use case, CAS reference equality would not work (sidestepping that JRuby pools some small range of Fixnums, I think it is from -127 to 127 or something).

This can be fixed by storing states in constants, this also improves readability:

```
class ReadWriteLock
  NOT_LOCKED   = 0
  READ_LOCKED  = 1
  WRITE_LOCKED = 2

  def initialize
    @state = NOT_LOCKED
  end

  def lock_for_read!
    cas_state(NOT_LOCKED, READ_LOCKED)
  end

  def lock_for_write!
    cas_state(NOT_LOCKED, WRITE_LOCKED)
  end

  def read_locked?
    @state == READ_LOCKED
  end

  def write_locked?
    @state == WRITE_LOCKED
  end
end
```

However this is Ruby and in Ruby we have symbols, that are guaranteed to be singletons:

```
class ReadWriteLock
  def initialize
    @state = :unlocked
  end

  def lock_for_read!
    cas_state(:unlocked, :read_locked)
  end

  def lock_for_write!
    cas_state(:unlocked, :write_locked)
  end

  def read_locked?
    @state == :read_locked
  end

  def write_locked?
    @state == :write_locked
  end
end
```

So in my opinion a pure raw-pointer/reference CAS implementation does not impede any of usual CAS use cases.

-thedarkone

#33 - 06/14/2014 08:56 PM - headius (Charles Nutter)

Vit Z wrote:

> In such a use case, CAS reference equality would not work (sidestepping that JRuby pools some small range of Fixnums, I think it is from -127 to 127 or something).

Increased to -256...255 at some point so we could represent an unsigned byte without construction. Otherwise, I agree with everything you said :-D

#34 - 04/08/2015 08:09 AM - hsbt (Hiroshi SHIBATA)

- Description updated

#35 - 01/25/2016 07:25 PM - pitr.ch (Petr Chalupa)

I would like to revive this issue again and link it to related efforts to provide complete set of low-level tools for writing concurrent libraries. The aggregating issue of this effort can be found here.

Summary can be found in this document: https://docs.google.com/document/d/1c07qfDArx0bhK9sMr24elaIUdOGudiqBhTIRALEbrYY/edit#. It

suggests to use following syntax attr :name, atomic: true, which makes the variable name volatile but it also generates atomic helpers: get_and_set_name, compare_and_set_name, compare_and_exchange_name, update_name. They use referential equality for comparison.

A version accessible without JS is here https://docs.google.com/document/d/1c07qfDArx0bhK9sMr24elaIUdOGudiqBhTIRALEbrYY/pub

#36 - 01/25/2016 10:19 PM - Eregon (Benoit Daloze)

- Related to Feature #12019: Better low-level support for writing concurrent libraries added

#37 - 01/26/2016 11:56 AM - pitr.ch (Petr Chalupa)

This could be implemented in MRI faster than on other platforms. While MRI has GIL it can do normal comparison and assignment without any synchronisation, instead of synchronised CAS operation, assuming that the implementation of the operation is written in C to prevent thread switching.

#38 - 12/23/2021 11:43 PM - hsbt (Hiroshi SHIBATA)

- Project changed from 14 to Ruby