Ruby - Feature #11539

Support explicit declaration of volatile instance variables

09/18/2015 09:47 PM - headius (Charles Nutter)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

SUMMARY

We seek to add a mechanism for explicitly declaring the volatility of a given instance variable and a formalization of current volatility semantics.

BACKGROUND

Concurrency is becoming more and more important in Ruby, and currently there are no guarantees that an instance variable is volatile.

Volatility in Java allows code to specify that reads and writes of a given field (Java's instance variables) should not be reordered with respect to each other. This is necessary to guarantee that concurrent reads and writes of those fields do not happen in unexpected orders, and it is a critical feature for building concurrency libraries in Java. Volatility is also used to provide hardware-level atomic libraries, a key component of lock-free data structures.

The Java memory model describes, among other things, how volatility works: http://www.cs.umd.edu/~pugh/java/memoryModel/

In CRuby, variables are implicitly volatile due to the fact that the GIL enforces memory barriers when switching threads. However, there is a growing desire to bring parallel threading to CRuby, and this will need to be addressed sooner rather than later.

In JRuby, this is a much bigger concern. Currently, JRuby uses volatile semantics for instance variable writes (not for reads) by default, because of the lack of a standard Ruby memory model. However, this forces undue performance constraints on all users: volatile instance variable writes, for example, have been measured to be 30% slower than non-volatile instance variable writes, and this is taking into account all the usual lookup, caching, and guard logic that surrounds them.

PURPOSE

This proposal seeks to formalize the fact that Ruby instance variables are not and have never been *guaranteed* to be volatile, and provide an explicit mechanism whereby users can opt-in to explicit volatility. This will provide immediate benefits in JRuby (reduced overhead of accessing instance variables) and eventual benefits for CRuby (explicit memory model surrounding instance variables for future parallel execution).

We believe that it should be possible to write efficient concurrent data structures in pure Ruby, and the first step toward this goal would be to have volatility as a first-class citizen of Ruby's object model.

PROPOSAL

1. Ruby instance variables would be explicitly defined as non-volatile by default.
2. A new method or syntax would be provided to explicitly opt-in to volatility. Some options:

FORM 1:

```
# This form builds off of existing modifer methods like Module#private.
volatile :foo
# No accessors would be created, leaving private variables private, but this would work too:
volatile attr_accessor :foo
```

FORM 2:

```
# This form overloads existing attr_accessor with keyword arguments, leaving room for other models
```

```
 like "atomic"
# Note that this form is less desirable since it requires creating accessor methods to get volatil
ity.
attr_accessor :foo, model: volatile
```

IMPLEMENTATION

The implementation in MRI would be minimal, given that instance variables are implicitly volatile right now. The new Module#volatile method would be a no-op in MRI until there's need for true volatility.

```
# Implememtation of volatile for current (non-parallel) MRI:
class Module
  def volatile(*syms)
  end
end
```

The implementation in JRuby would be trivial, since we already have logic in place to make instance variable reads and writes volatile. We'd simply make them non-volatile globally and add this method to turn volatility on.

So basically the work is already done in both implementations (other than wiring up the Module#volatile method).

RISKS

There is risk that the new Module#volatile would conflict with some library's addition to Module of the same name, or conflict with some library's metaclass-decorating module that provides a method of the same name. However, we do not believe this is a high risk. We have not done any searches at this time to see if there are such conflicts.

NAMING

We believe volatile is the best name for this feature, since it is the accepted term in most programming communities (C++ register volatility, Java and C# volatile fields, etc).

TIMEFRAME

As soon as possible. We would like to see this in Ruby 2.3 ideally. Given that it is a no-op in MRI and nearly available in JRuby, this should be feasible.

RELATED WORK

See Java, C# as examples of volatile in real-world practice. Java has an extensive set of lock-free concurrent data structures built entirely in Java atop the ability to declare field volatility.

The concurrent-ruby project (https://github.com/ruby-concurrency/concurrent-ruby) implements this feature as a separate class hierarchy, Synchronization::Object. Classes descending from this hierarchy gain the ability to request construction-time memory fencing and instance variable volatility. While this approach works, we believe it would be better to have as a core Ruby feature that can be applied to any instance variable in any class.

C++ also has a notion of volatility as a compiler hint to not reorder stores and loads of a memory location: http://en.cppreference.com/w/cpp/language/cv

Related issues:

Related to Ruby - Feature #12019: Better low-level support for writing concur...	**Assigned**
Related to Ruby - Feature #6470: Make attr_accessor return the list of genera...	**Closed**

History

#1 - 09/18/2015 09:53 PM - headius (Charles Nutter)

If someone can fix the formatting I'd appreciate it. I don't know how.

#2 - 09/19/2015 12:39 AM - nobu (Nobuyoshi Nakada)

- Description updated

#3 - 09/20/2015 01:37 AM - duerst (Martin Dürst)

Charles Nutter wrote:

> # RISKS
>
> There is risk that the new Module#volatile would conflict with some library's addition to Module of the same name, or conflict with some library's metaclass-decorating module that provides a method of the same name. However, we do not believe this is a high risk. We have not done any searches at this time to see if there are such conflicts.

I think the main risk of this proposal is that it's very easy to implement now (especially for MRI), but it's unclear what costs it might create in the future. We should try to get an idea of this before we commit to this proposal.

As an example of what I mean, it would be bad if e.g. this proposal would complicate and therefore delay other work on concurrency in MRI.

#4 - 09/21/2015 12:00 PM - headius (Charles Nutter)

The costs for MRI would be minimal even in the future, because this is a very conservative approach to providing visibility and ordering guarantees (and the same as the other big managed languages that have parallel threading).

In MRI, if there comes a time when parallel threads are available, the additional work to make certain instance variables volatile will be no more than having a flag in the instance variable table to indicate that a memory fence should be inserted to guarantee store/load ordering. We are doing this already in JRuby by emitting an x86 LOCK ADDL (via JVM instrinsic).

The cost of making everything be volatile all the time is pretty significant too: in the neighborhood of 1.5x to 3.0x more cycles per write on x86, and probably more on platforms that do not have automatic cache coherency like ARM. That's a big reason why we believe now is the time to make a firm statement on how parallel threads and Ruby instance variables should work.

Another resource I forgot to mention is the JRuby wiki page on concurrency: https://github.com/jruby/jruby/wiki/Concurrency-in-jruby

In order to help users write better parallel code on JRuby, we have tried to explicitly spell out thread-related guarantees for various types of variables. Note that class variables, constant tables, method tables, thread-local variables, and global variables are all already implemented in a volatile way because of the nature of their more global usage.

#5 - 01/25/2016 07:23 PM - pitr.ch (Petr Chalupa)

Linking this issue with 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 the following syntax attr :name, volatile: true, which makes @name and @name = 'value' a volatile read and a volatile write. If attr_(reader|writer) is called in class in addition to attr it creates volatile reader and volatile writer methods.

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

#6 - 01/25/2016 10:19 PM - Eregon (Benoit Daloze)

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

#7 - 01/10/2019 09:16 AM - Eregon (Benoit Daloze)

- Related to Feature #6470: Make attr_accessor return the list of generated method added

#8 - 01/26/2021 03:30 PM - Eregon (Benoit Daloze)

- Description updated