Ruby Language Optimization Techniques

NICHOLAS BENDER, BOISE STATE UNIVERSITY
JOHN OTANDER, BOISE STATE UNIVERSITY
BEN NEELY, BOISE STATE UNIVERSITY

The Ruby programming language has experienced a recent period of intense adoption and growth due to its excellent speed of iteration and due in no small part to the acceptance of the Ruby on Rails web framework within the startup sphere. While support is growing steadily for the language, it is largely dismissed as not having effective scalability, or having far slower runtimes than more traditional strongly-typed complex languages. In this article, we propose that many sophisticated techniques exist to enhance Ruby's performance both in using existing runtimes to compile ruby to statically typed languages, and in using common anti-patterns to improve performance natively. Through experimentation and thorough research we conclude that Ruby performs competitively against it's similar scripting language counterparts, and can see increases of [XXXXX]% in many cases.

Categories and Subject Descriptors: **D.2.3 [Coding Tools and Techniques]**: Object-oriented programming, **B.6.3 [Design Aids]: Optimization**

General Terms: Optimization, Algorithms, Performance

Additional Key Words and Phrases: Ruby, Web Development, JRE, C++,

ACM Reference Format:

Gang Zhou, Yafeng Wu, Ting Yan, Tian He, Chengdu Huang, John A. Stankovic, and Tarek F. Abdelzaher, 2010. A multi-frequency MAC specially designed for wireless sensor network applications. *ACM Trans. Embedd.Comput.Syst.*9,4,Article39(March2010),6pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In recent years, the Ruby programming language has grown its community and established itself as a valuable and popular tool for many tasks [O'Donoghue, 2014]. The success of Ruby on Rails as a prototyping framework as well as a full-stack solution for some larger companies has brought forth a myriad of techniques to ensure that the language's speed differences compared to similar languages are minimal. Ruby's slower performance as compared to C or Java is attributed to interpreted execution, dynamic typing, meta-programming support, and the Global Interpreter Lock [Odaira, Castanos, and Tomari, 2014]. This increase in popularity has caused a large number independent optimization efforts to arise from large corporations such as IBM, as well as efforts from the Ruby open-source community.

Author's addresses: N. Bender, Boise State University, Computer Science Department, B Neely, Boise State University, Computer Science Department, J. Otander, Boise State University, Computer Science Department.

DOI:http://dx.doi.org/10.1145/0000000.0000000

With each of these techniques there exist certain sacrifices, but in this exploration we will conclude that the best practices for stable, performant Ruby code exist by utilizing the newest versions of the core language properly, and not by utilizing other third party interpreters or solutions.

Ruby is an object oriented, dynamically-typed, high-level scripting language. It is a programming language that was written for humans and just happens to run on computers. It's intended to promote developer happiness through simplicity, elegant libraries, and terse, readable syntax. Ruby uses duck typing, meaning type is determined through methods and properties.

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
Heim, Michael (2007). Exploring Indiana Highways. Exploring America's Highway. p. 68. ISBN 978-0-9744358-3-1.

Program Execution at a High Level

CODE => TOKENIZATION => PARSE TREE => COMPILATION => YARV INSTRUCTIONS

When a Ruby program is executed, it first tokenizes the program. This means that the contents are converted into a collection of tokens with associated types. Ruby uses the LALR (Look-Ahead Left Reversed Rightmost Derivation) Parser to apply meaning to the tokens and construct the Abstract Syntax Tree. The compilation step was introduced with Ruby 1.9, and is where the YARV (Yet Another Ruby Virtual Machine) comes into play. It translates the code into bytecode, or YARV instructions.

YARV instructions for a simple program:

```
~|||$ irb
2.1.1 :001 > code = <<CODE
2.1.1 :002"> puts 1 + 2
2.1.1 :003"> CODE
 => "puts 1 + 2\n"
2.1.1 :004 > puts RubyVM::InstructionSequence.compile(code).dis-
asm
== d i s a s m :
<RubyVM::InstructionSequence:<compiled>@<compiled>>==========
0 0 0 0  t r a c e                                          1
(   1)
0002 putself
0003 putobject_OP_INT2FIX_O_1_C_
0004 putobject        2
0006 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
0008  opt_send_simple    <callinfo!mid:puts, argc:1, FCALL|
ARGS_SKIP>
0010 leave
 => nil
```

The introduction of the compilation step and YARV have significantly helped the execution speed of Ruby programs. However, there's always room for more improvements.

2. JRUBY

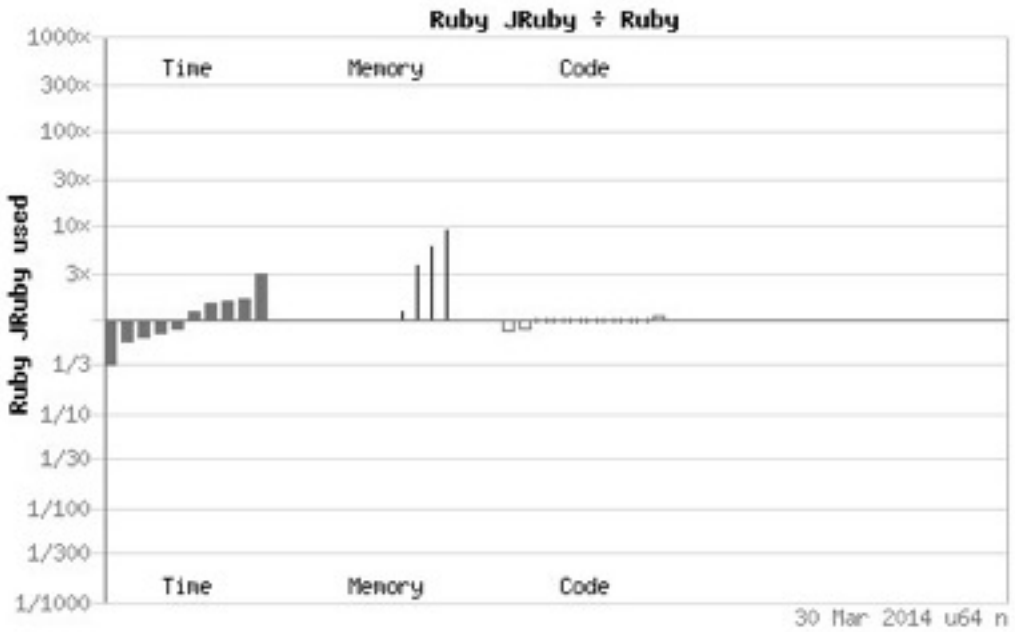
1. Purpose

Jruby endeavors to solve many Ruby performance issues by eliminating the standard interpreter and instead taking ruby syntax and compiling as much of the core libraries as possible to Java bytecode. Current versions of JRuby support both just-in-time compilation as well as ahead-of-time compilation to Java bytecode. In using these various stages of bytecode in addition to some portions of the standard interpreter, this allows for several advantages over the standard interpreter.

One of the more obvious improvements is the ability to call and use standard Java libraries and classes from within ruby projects. For larger organizations already using Java for core library support, this allows for improved flexibility of the development environment.

2.2. Performance

In 2007, JRuby's overall performace was compared with Ruby 1.8.5, the Yarv interpreter (now merged into Ruby's official interpreter), and Rubinius. In it, only 10% of tests performed had JRuby outperforming standard Ruby. These speed enhancements, however, still managed to run all Ruby benchmarks without timing out or producing an error, a claim that no other non-standard Ruby implementation could make [Cangiano, 2007].



(Figure 1)

However, benchmarks performed in 2014 between the latest implementations of JRuby and Ruby (Figure 1) are comparable to standard Ruby, but also ramp up significantly in comparison to Ruby in Memory usage (to almost 10x).

If memory usage isn't a priority for a given Ruby project, the biggest additional downside in performance of JRuby has to do with the speed of initializing the JVM to begin with. A simple ruby script that would take the MRI a fraction of a second to run would require several additional seconds just due to JVM launch times.

2.3 Lack of C Support

While JRuby allows for enhanced support and compatibility with Java libraries and applets, the majority of Ruby users (especially those using Ruby on Rails) are used to using libraries that contain native C support. In choosing to support Java, JRuby forces the incompatibility with native C extensions. Most notably are a variety of database interfaces and web servers.

2.4. Development Lag

Due to JRuby's implementation being dependent on Ruby releases prior to implementation and support, this has created an unfortunately long lag time, with the most recent release of JRuby only supporting Ruby version 1.9.3, which was initially released in 2011.

2.5 Summary

While JRuby does offer some improved benchmark performance in a minority of cases, the slow development cycle and potential for a massive increase to memory footprint make it an unsuitable option for pure ruby development stacks.

3. RUBINIUS

1. Purpose

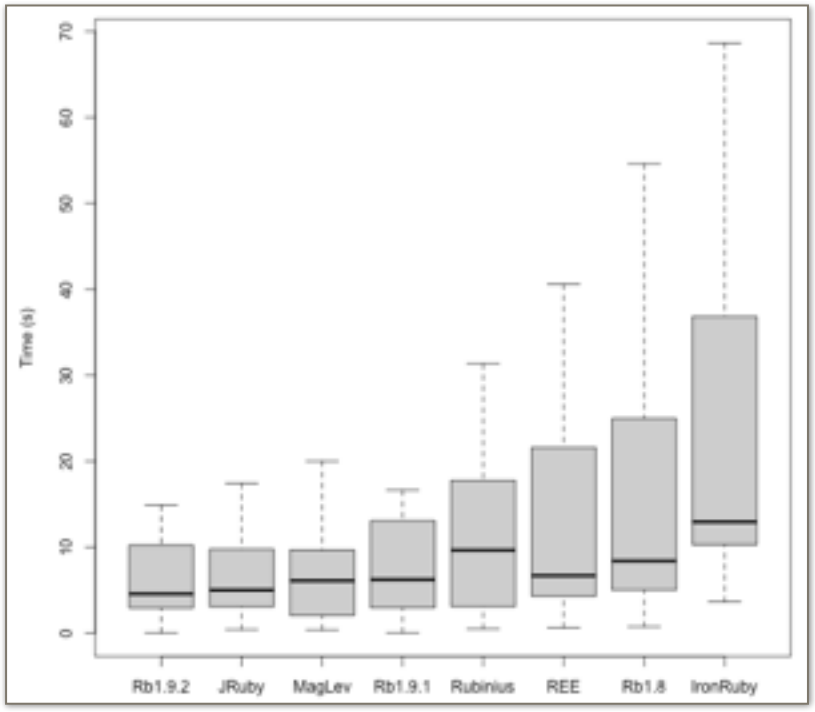
Rubinius is an implementation of the Ruby programming language and includes a bytecode virtual machine, Ruby syntax parser, bytecode compiler, generational garbage collector, just-in-time (JIT) native machine code compliler, and Ruby Core and Standard Libraries. Rubinius is written using Ruby and C++.

2. History

Rubinius was originally created to be a Ruby virtual machine and runtime written in pure ruby. The current ruby interpreter is primarily writen in non-Ruby langauges such as C. From 2007 to 2013, the software company Engine Yard was a primary backer of Rubinius. During that time the focus of Rubinius evolved from creating a completely bootstrapped Ruby VM to instead offering an implementation of Ruby with increased performance. Under this new direction, Rubinius partially abandoned the idea of bootstrapping the Ruby VM in all Ruby code, and instead sought to use C ++ to increase performance and establish Rubinius as the fastest Ruby implementation. Recently Rubinius has focused on supporting concurrency and multi-threading.

3. Performance

Rubinius initially achieved performance equal or slightly better to that of the Yarv interpreter. However, in recent years the MRI interpreter has consistently out performed Rubinius on most benchmark tests.



Rubinius consistently benchmarks as one of the slowest modern implementations of the Ruby language.

3.4 Concurrency

Rubinius does outperform the MRI in threading and concurrency benchmark tests. As shown in the figure bellow, Rubinius (represented by rbx-2.0.0) has a nontrivial advantage over MRI and other Ruby implementations when exciting multithreaded code.

Rubinius is unique amongst Ruby implementations in that it does not have Global Interpreter Lock (GIL). The GIL in all other Reuby implementation allows only one

Benchmark File	Input Size	mri 1.9.3	mri 2.0.0	mri 2.1.0dev	jruby-1.7.4	rbx-2.0.0
micro/bm_count_multithreaded.rb	1	0.005267	0.005189	0.005413	0.007000	0.001949
micro/bm_count_multithreaded.rb	2	0.010460	0.010471	0.011105	0.009000	0.002033
micro/bm_count_multithreaded.rb	4	0.020324	0.021424	0.021594	0.014000	0.005141
micro/bm_count_multithreaded.rb	8	0.043974	0.043954	0.048571	0.023000	0.008579
micro/bm_count_multithreaded.rb	16	0.095841	0.095974	0.093919	0.028000	0.017421
micro/bm_count_shared_thread.rb	1	0.050200	0.050026	0.052267	0.054000	0.016658
micro/bm_count_shared_thread.rb	2	0.053162	0.052976	0.055364	0.035000	0.008967
micro/bm_count_shared_thread.rb	4	0.054265	0.053976	0.055196	0.034000	0.008960
micro/bm_count_shared_thread.rb	8	0.056671	0.054181	0.059954	0.015000	0.010952
micro/bm_count_shared_thread.rb	16	0.051070	0.054593	0.053923	0.020000	0.011171

thread to execute at at a time, no matter how many processor cores are available. Not implementing the GIL gives Rubinius the ability to support true threading

3.5 Conclusion

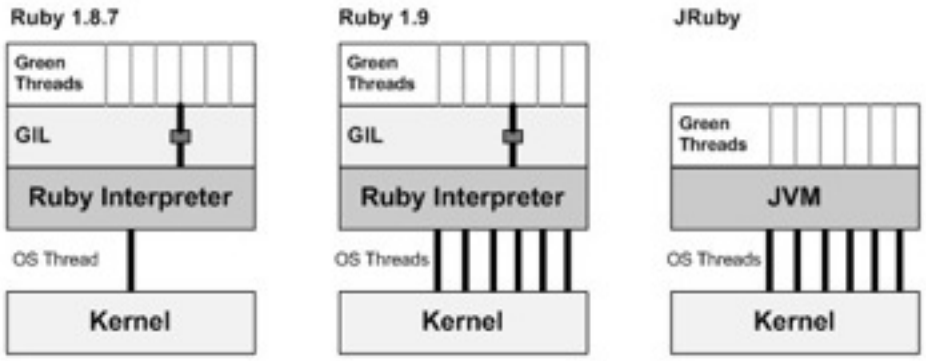
Rubinius' development has been spotty, depending heaving on a few developers and a few corporate sponsors. As a result Rubinius has constantly shifted focus. Rubinius currently offers a significant advantage over other Ruby interpreters only with regards to programming involving threading and concurrency. For all other uses, the standard MRI Ruby interpreter is faster and more consistently supported.

4. MRI/YARV

4.1 Purpose

The Ruby MRI is short for Matz's Ruby Interpreter, and is the reference implementation for the Ruby programming language. It was released to the public in 1995, and is still actively developed, with the latest stable build being Ruby 2.1.1.

The YARV is an interpreter developed by Koichi Sasada that's also known as the



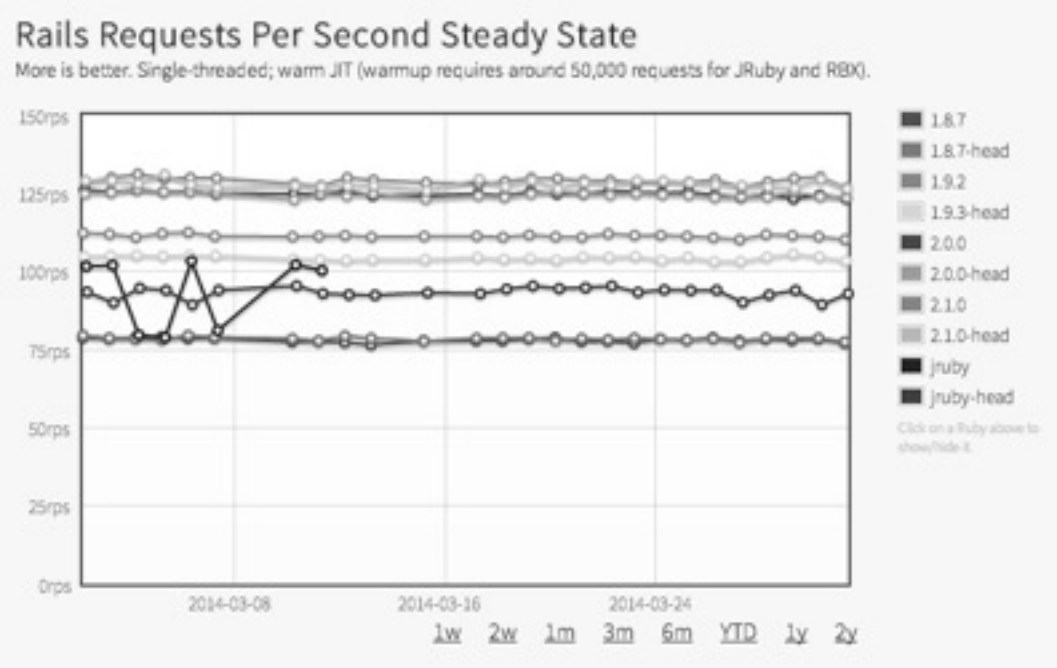
KRI. It was developed in order to reduce the execution time of Ruby programs, and was very successful. As a result, YARV was merged into Ruby 1.9.0 and has replaced the MRI.

As the default interpreter for the Ruby programming language, the MRI has received it's fair share of criticism, primarily due to it's execution speeds and memory consumption. However, recent Ruby versions have seen significant enhancements, and is on par with similar scripting languages like Python. Not to mention the fact that some comparisons have the audacity to compare a compiled language to a scripting language, which is apples to oranges. Developers typically choose Ruby for it's ease of writing/prototyping, understanding the fact that its execution time will always be significantly slower than its compiled counterparts.

That being said, there are numerous methods and best practices that developers can follow in order to ensure that they're avoiding unnecessary bottlenecks.

4.2 Performance Out of the Box

Thanks to the introduction of YARV, vanilla Ruby, on a single thread, has the ability to outperform other alternative Ruby implementations. Consider the following figure, that measures Rails requests per second.



4.3 Global Interpreter Lock

When attempting to optimize execution speed, threads are often utilized in order to process tasks concurrently. This is a feature that Ruby supports, too. However, the MRI/YARV incorporates a Global Interpreter Lock, or GIL, that doesn't permit any true concurrency. A GIL refers to an interpreter thread that doesn't allow code that isn't thread safe to share itself with other threads. This results in little, to no, actual gain in speed when running threads on a multiprocessor machine.

The primary reason is that the GIL is used to avoid race conditions within C extensions. There are also thread safety reasons, too. Parts of Ruby aren't thread safe (Hash), and numerous C libraries that are wrapped by Ruby's internals. Additionally, the GIL is integral to data integrity, because it ensures that the developer doesn't write any unsafe threading code.

This, interestingly enough, runs contrary to the fundamental principles of the Ruby language, where all the responsibility is laid on the developer. Ruby allows the developer to have the ultimate freedom without hand holding, yet the GIL is just that, hand holding.

The GIL isn't going anywhere. It is deeply intertwined with Ruby and its internals, and many influential Ruby-core figures don't plan on removing the GIL anytime in the near future. Though, this doesn't mean the concurrency can't be achieved.

Though, you can sidestep the GIL with multiple virtual machines. Sasada Koichi has proposed a Multiple VM (MVM) solution, which is currently being developed. This would consist of multiple virtual machines, running their own processes, and communicate via sockets.

Granted, this is a drastic step away from typical threading, but some proponents believe that traditional threading isn't necessarily the correct paradigm to follow. Especially considering the fact the Ruby leverages green threads above the GIL rather than talking to the OS directly.

Nevertheless, you're right the GIL is not as bad as you would initially think: you just have to undo the brainwashing you got from Windows and Java proponents who seem to consider threads as the only way to approach concurrent activities. Just because Java was once aimed at a set-top box OS that didn't support multiple address spaces, and just because process creation in Windows used to be slow as a dog, doesn't mean that multiple processes (with judicious use of IPC) aren't a much better approach to writing apps for multi-CPU boxes than threads.

4.4 Simple Code Enhancements

String interpolation is significantly more performant than concatentation because it doesn't need to allocate new strings, it just modifies a single string in place.

```
require 'benchmark'

concat_time = Benchmark.measure do
  20000000.times do
    str = 'str1' << 'str2' << 'str3'
  end
end

# => #<Benchmark::Tms:0x007fdf9ba49ea8 @label="",
@real=79.152523, @cstime=0.0, @cutime=0.0,
@stime=0.04000000000000001, @utime=79.11, @total=79.15>

interp_time = Benchmark.measure do
  20000000.times do
    str = 'str1' << 'str2' << 'str3'
  end
end

# => #<Benchmark::Tms:0x007fdf9b990bd8 @label="",
@real=22.713976, @cstime=0.0, @cutime=0.0,
@stime=0.009999999999999995, @utime=22.689999999999998,
@total=22.7>
```

The `collect|map` methods with blocks are faster because it returns a new array rather than an enumerator. This can be leveraged to increase speed when compared to `Symbol.to_proc` implementations. Though, the latter is typically much more preferable to read. The reason that the Symbol.to_proc is slower is because to_proc is called on the symbol to perform the following conversion:

```
:method.to_proc
# => -> x { x.method }
fake_data = 20.times.map { |t| Fake.new(t) }

proc_time = Benchmark.measure do
  200000000.times do
    fake_data.map(&:id)
  end
end

#     => #<Benchmark::Tms:0x007fdf9b8b0498 @label="",
@real=491.332415, @cstime=0.0, @cutime=0.0, @stime=4.8,
@utime=426.06999999999994, @total=430.86999999999995>

block_time = Benchmark.measure do
  200000000.times do
    fake_data.map { |d| d.id }
  end
end

#  => #<Benchmark::Tms:0x007fdf9b931d40 @label="",
@real=431.731424, @cstime=0.0, @cutime=0.0, @stime=2.66,
@utime=416.21000000000004, @total=418.87000000000006>

collect_time = Benchmark.measure do
  200000000.times do
    fake_data.collect { |d| d.id }
  end
end

#  => #<Benchmark::Tms:0x007fdf9b821518 @label="",
@real=386.234513, @cstime=0.0, @cutime=0.0,
@stime=1.1800000000000006, @utime=384.28, @total=385.46>
 :037 >
```

There are also garbage collection modifications that can be made in order to further optimize Ruby execution speed for most systems.

```
# This is 60(!) times larger than default
RUBY_HEAP_MIN_SLOTS=600000

# This is 7 times larger than default
RUBY_GC_MALLOC_LIMIT=59000000

# This is 24 times larger than default
RUBY_HEAP_FREE_MIN=100000
```

4.4 Use Unicorn

For Ruby on Rails web applications, a server typically runs on a single process, which means that every request is processed one at a time. This can create a significant bottle neck in your application. Fortunately, there are libraries to incorporate concurrency in your application. One of which is Unicorn.

Unicorn uses Unix forks within a dyno (web worker) to create multiple instances of itself. Now, there are multiple OS instances that can all respond to requests, and complete tasks concurrently. This results in smaller queues, quicker responses, and a faster web application as a whole. The only drawback is memory usage, which can grow to large sizes. Though, with decreasing hardware costs, this becomes a worthwhile expenditure to ensure quick development time for the software components. This also doesn't require thread safe code, since each worker is a self-sufficient clone of the parent.

Ruby 2.0 makes process forking even more efficient with Unicorn because it implements Copy-on-Write (CoW), which means that a parent and child share physical memory until a write needs to be made. This is a very efficient sharing of resources that can drastically reduce memory use.

Sometimes, there are still issues with memory leakage, which occurs when workers get stuck or timeout. With the inclusion of a gem, and a small snippet of code that's included below, these edge cases are covered.

```
# --- Start of unicorn worker killer code ---

if ENV['RAILS_ENV'] == 'production'
  require 'unicorn/worker_killer'

  max_request_min =  500
  max_request_max =  600

  # Max requests per worker
    use  Unicorn::WorkerKiller::MaxRequests,  max_request_min,
max_request_max

  oom_min = (240) * (1024**2)
  oom_max = (260) * (1024**2)

  # Max memory size (RSS) per worker
  use Unicorn::WorkerKiller::Oom, oom_min, oom_max
end

# --- End of unicorn worker killer code ---

require ::File.expand_path('../config/environment',  __FILE__)
run YourApp::Application
```

CONCLUSIONS

In this article, we examined a number independent Ruby optimization efforts. Each of these efforts seek to achieve performance improvements through a variety of techniques. In our examination we've determined that for each of these techniques there

are certain sacrifices, that outweigh the marginal benefits are gained. Unless a particular feature is needed (such as full threading support or inline Java) the best practices for stable, performant Ruby code exist by utilizing the newest versions of the core language.

6. TYPICAL REFERENCES IN NEW ACM REFERENCE FORMAT

A paginated journal article [Abril and Plant 2007], an enumerated journal article [Cohen et al. 2007], a reference to an entire issue [Cohen 1996], a monograph (whole book) [Kosiur 2001], a monograph/whole book in a series (see 2a in spec. document) [Harel 1979], a divisible-book such as an anthology or compilation [Editor 2007] followed by the same example, however we only output the series if the volume number is given [Editor 2008] (so Editor00a's series should NOT be present since it has no vol. no.), a chapter in a divisible book [Spector 1990], a chapter in a divisible book in a series [Douglass et al. 1998], a multi-volume work as book [Knuth 1997], an article in a proceedings (of a conference, symposium, workshop for example) (paginated proceedings article) [Andler 1979], a proceedings article with all possible elements [Smith 2010], an example of an enumerated proceedings article [Gundy et al. 2007], an informally published work [Harel 1978], a doctoral dissertation [Clarkson 1985], a master's thesis: [Anisi 2003], an online document / world wide web resource [Thornburg 2001], [Ablamowicz and Fauser 2007], [Poker-Edge.Com 2006], a video game (Case 1) [Obama 2008] and (Case 2) [Novak 2003] and [Lee 2005] and (Case 3) a patent Scientist 2009], work accepted for publication [Rous 2008], 'YYYYb'-test for prolific author [Saeedi et al. 2010a] and [Saeedi et al. 2010b]. Other cites might contain 'duplicate' DOI and URLs (some SIAM articles) [Kirschmer and Voight 2010]. Boris / Barbara Beeton: multi-volume works as books [Hörmander 1985b] and [Hörmander 1985a].

ACKNOWLEDGMENTS

The authors would like to thank Douglas Wiegley. He knows what he did.

REFERENCES

Robert O'Donoghue. 2014. Careers Close-up: programmers and software engineers. (March 2014). Retrieved March 31, 2014 http://www.siliconrepublic.com/careers/item/36001-crs-cls-up

Rei Odaira, Jose G. Castanos, Hisanobu Tomari, 2014, *Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory*. PPoPP'14, February 15-19 2014, Orlando, FL, USA. DOI: http://dx.doi.org/10.1145/2555243.2555247

Antonio Cangiano. 2007. The Great Ruby Shootout (December 2007). Retrieved March 31, 2014 http://programmingzen.com/2007/12/03/the-great-ruby-shootout/

Sten Andler. 1979. Predicate Path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226–236. DOI:http://dx.doi.org/10.1145/567752.567774

David A. Anisi. 2003. *Optimal Motion Control of a Ground Vehicle*. Master's thesis. Royal Institute of Technology (KTH), Stockholm, Sweden.

Brian Cabral and Leith C. Leedom. 1993. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'93)*. ACM, New York, NY, 263–270. DOI:http://dx.doi.org/10.1145/166117.166151

Kenneth L. Clarkson. 1985. *Algorithms for Closest-Point Problems (Computational Geometry)*. Ph.D. Dissertation. Stanford University, Palo Alto, CA. UMI Order Number: AAT 8506171.

Jacques Cohen (Ed.). 1996. Special Issue: Digital Libraries. *Commun. ACM* 39, 11 (Nov. 1996).

Sarah Cohen, Werner Nutt, and Yehoshua Sagic. 2007. Deciding equivalances among conjunctive aggregate queries. *J. ACM* 54, 2, Article 5 (April 2007), 50 pages. DOI:http://dx.doi.org/10.1145/1219092.1219093

John G. Daugman. 1985. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two dimensional visual cortical filters. *J. Optical Soc. Amer. A: Optics, Image Science, Vision* 2, 7 (1985), 1160–1169.

Bruce P. Douglass, David Harel, and Mark B. Trakhtenbrot. 1998. Statecarts in use: structured analysis and object-orientation. In *Lectures on Embedded Systems*, Grzegorz Rozenberg and Frits W. Vaandrager (Eds.). Lecture Notes in Computer Science, Vol. 1494. Springer-Verlag, London, 368–394. DOI:http://dx.doi.org/10.1007/3-540-65193-4 29

Ian Editor (Ed.). 2007. *The title of book one* (1st. ed.). The name of the series one, Vol. 9. University of Chicago Press, Chicago. DOI:http://dx.doi.org/10.1007/3-540-09237-4

Ian Editor (Ed.). 2008. *The title of book two* (2nd. ed.). University of Chicago Press, Chicago, Chapter 100. DOI:http://dx.doi.org/10.1007/3-540-09237-4

David J. Field, Anthony Hayes, and Robert F. Hess. 1993. Contour integration by the human visual system: Evidence for a local "association field". *Vision Res*. 33, 2 (1993), 173–193. DOI:http://dx.doi.org/10.1016/0042-6989(93)90156-Q

David Fowler and Colin Ware. 1989. Strokes for Representing Univariate Vector Field Maps. In *Proceedings of Graphics Interface*. Canadian Human-Computer Communications Society, Mississauga, Ontario, 249–253.

Matthew Van Gundy, Davide Balzarotti, and Giovanni Vigna. 2007. Catch me, if you can: Evading network signatures with web-based polymorphic worms. In *Proceedings of the first USENIX workshop on Offensive Technologies (WOOT '07)*. USENIX Association, Berkley, CA, Article 7, 9 pages.

David Harel. 1978. *LOGICS of Programs: AXIOMATICS and DESCRIPTIVE POWER*. MIT Research Lab Technical Report TR-200. Massachusetts Institute of Technology, Cambridge, MA.

David Harel. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, Vol. 68. Springer-Verlag, New York, NY. DOI:http://dx.doi.org/10.1007/3-540-09237-4

Lars Hörmander. 1985a. *The analysis of linear partial differential operators. III*. Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], Vol. 275. Springer-Verlag, Berlin, Germany. viii+525 pages. Pseudodifferential operators.

Lars Hörmander. 1985b. *The analysis of linear partial differential operators. IV*. Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], Vol. 275. Springer-Verlag, Berlin, Germany. vii+352 pages. Fourier integral operators.

David H. Hubel and Torsten N. Wiesel. 1962. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J. Physiol*. 160, 1 (1962), 106–154. http://jp.physoc.org

David H. Hubel and Torsten N. Wiesel. 1968. Receptive fields and functional architecture of monkey striatecortex.(1968).http://jp.physoc.org/cgi/content/abstract/195/1/215 http://www.hubel/papers/uconn.html.

Bruno Jobard and Wilfrid Lefer. 1997. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings of the Eurographics Workshop*. Springer Verlag, Berlin, 43–56.

Markus Kirschmer and John Voight. 2010. Algorithmic Enumeration of Ideal Classes for Quaternion Orders. *SIAM J. Comput*. 39, 5 (Jan. 2010), 1714–1747. DOI:http://dx.doi.org/10.1137/080734467

Donald E. Knuth. 1997. The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd. ed.). Addison Wesley Longman Publishing Co., Inc.

David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY.

David H. Laidlaw, J. Scott Davidson, Timothy S. Miller, Marco da Silva, R. M. Kirby, William H. Warren, and Michael Tarr. 2001. Quantitative comparative evaluation of 2D vector field visualization methods. In *Proceedings of the Conference on Visualization (VIS'01)*. IEEE Computer Society, Los Alamitos, CA, 143–150.

Newton Lee. 2005. Interview with Bill Kinder: January 13, 2005. Video, *Comput. Entertain*. 3, 1, Article 4 (Jan.-March 2005). DOI:http://dx.doi.org/10.1145/1057270.1057278

Zhaoping Li. 1998. A neural model of contour integration in the primary visual cortex. *Neural Comput*. 10, 4 (1998), 903–940. DOI:http://dx.doi.org/10.1162/089976698300017557

Nick Lund. 2001. *Attention and Pattern Recognition*. Routledge, New York, NY.

Dave Novak. 2003. Solder man. Video. In *ACM SIGGRAPH 2003 Video Review on Animation theater Program: Part I - Vol. 145 (July 27–27, 2003)*. ACM Press, New York, NY, 4. DOI:http://dx.doi.org/99.9999/woot07-S422

Barack Obama. 2008. A more perfect union. Video. (5 March 2008). Retrieved March 21, 2008 from http://video.google.com/videoplay?docid=6528042696351994555

Daniel Pineo and Colin Ware. 2008. Neural modeling of flow rendering effectiveness. In *Proceedings of the 5th Symposium on Applied Perception in Graphics and Visualization (APGV'08)*. ACM, New York, NY, 171–178. DOI:http://dx.doi.org/10.1145/1394281.1394313

Poker-Edge.Com. 2006. Stats and Analysis. (March 2006). Retrieved June 7, 2006 from http://www.poker-

edge.com/stats.php

Bernard Rous. 2008. The Enabling of Digital Libraries. *Digital Libraries* 12, 3, Article 5 (July 2008). To appear.

Mehdi Saeedi, Morteza Saheb Zamani, and Mehdi Sedighi. 2010a. A library-based synthesis methodology for reversible logic. *Microelectron*. J. 41, 4 (April 2010), 185–194.

Mehdi Saeedi, Morteza Saheb Zamani, Mehdi Sedighi, and Zahra Sasanian. 2010b. Synthesis of Reversible Circuit Using Cycle-Based Approach. *J. Emerg. Technol. Comput. Syst.* 6, 4 (Dec. 2010).

Joseph Scientist. 2009. The fountain of youth. (Aug. 2009). Patent No. 12345, Filed July 1st., 2008, Issued Aug. 9th., 2009.

Stan W. Smith. 2010. An experiment in bibliographic mark-up: Parsing metadata for XML export. In *Proceedings of the 3rd. annual workshop on Librarians and Computers (LAC '10),* Reginald N. Smythe and Alexander Noble (Eds.), Vol. 3. Paparazzi Press, Milan Italy, 422–431. DOI:http://dx.doi.org/99.9999/woot07-S422

Asad Z. Spector. 1990. Achieving application requirements. In *Distributed Systems* (2nd. ed.), Sape Mullender (Ed.). ACM Press, New York, NY, 19–33. DOI:http://dx.doi.org/10.1145/90417.90738

Harry Thornburg. 2001. Introduction to Bayesian Statistics. (March 2001). Retrieved March 2, 2005 from http://ccrma.stanford.edu/~jos/bayes/bayes.html

Greg Turk and David Banks. 1996. *Image-guided streamline placement*. Technical Report I-CA2200. UniversityofCalifornia,SantaBarbara,CA.453–460pages. DOI:http://dx.doi.org/10.1145/237170.237285

Colin Ware. 2008. Toward a Perceptual Theory of Flow Visualization. *IEEE Comput. Graph. Appl.* 28, 2 (2008), 6–11. DOI:http://dx.doi.org/10.1109/MCG.2008.39

Sources to be formatted:

GIL
https://mail.python.org/pipermail/python-3000/2007-May/007414.html
http://archive.is/yCFB
http://archive.is/X1kh
http://www.confreaks.com/videos/1272-rubyconf2012-implementation-details-of-ruby-2-0-vm
https://news.ycombinator.com/item?id=3070382
http://merbist.com/2011/10/18/data-safety-and-gil-removal/
http://merbist.com/2011/10/03/about-concurrency-and-the-gil/
GC
http://www.rubyenterpriseedition.com/documentation.html
https://lightyearsoftware.com/2012/11/speed-up-mri-ruby-1-9/
Unicorn
https://www.digitalocean.com/community/articles/how-to-optimize-unicorn-workers-in-a-ruby-on-rails-app
Use Ruby Threads and Fibers
http://merbist.com/2011/02/22/concurrency-in-ruby-explained/
Fine Tune Your Objects
http://patshaughnessy.net/2013/2/8/ruby-mri-source-code-idioms-3-embedded-objects
Code optimizations
http://www.ruby-doc.org/core-2.1.1/Array.html#M000249

Received February 2007; revised March 2009; accepted June 2009

Online Appendix to: A Multifrequency MAC Specially Designed for Wireless Sensor Network Applications

GANG ZHOU, COLLEGE OF WILLIAM AND MARY
YAFENG WU, UNIVERSITY OF VIRGINIA
TING YAN, EATON INNOVATION CENTER
TIAN HE, UNIVERSITY OF MINNESOTA
CHENGDU HUANG, GOOGLE
JOHN A. STANKOVIC, UNIVERSITY OF VIRGINIA
TAREK F. ABDELZAHER, UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

A. THIS IS AN EXAMPLE OF APPENDIX SECTION HEAD

Channel-switching time is measured as the time length it takes for motes to successfully switch from one channel to another. This parameter impacts the maximum network throughput, because motes cannot receive or send any packet during this period of time, and it also affects the efficiency of toggle snooping in MMSN, where motes need to sense through channels rapidly.

By repeating experiments 100 times, we get the average channel-switching time of Micaz motes: 24.3 μs. We then conduct the same experiments with different Micaz motes, as well as experiments with the transmitter switching from Channel 11 to other channels. In both scenarios, the channel-switching time does not have obvious changes. (In our experiments, all values are in the range of 23.6 μs to 24.9 μs.)

B. APPENDIX SECTION HEAD

The primary consumer of energy in WSNs is idle listening. The key to reduce idle listening is executing low duty-cycle on nodes. Two primary approaches are considered in controlling duty-cycles in the MAC layer.

DOI:http://dx.doi.org/10.1145/0000000.0000000