rubybib.org

The Ruby Bibliography

The Ruby programming language hasn't historically been the subject of much research, either in industry or academia. A lot of recent systems research has used languages like C, C++ and Java. Contemporary programming language research often uses languages like Java, Scala and Haskell. Modern research into VMs, compilers and garbage collectors is often based on Java or recently Python.

However there are now a growing number of research projects using Ruby. On this page we list theses and peer-reviewed papers and articles that cover Ruby implementation or use Ruby, including alternative implementations such as JRuby. We also list other notable papers that don't mention Ruby but are applied in Ruby implementations.

All paper links were publicly available.

Ruby

Virtual Machines

- M. Viering. **An Efficient Runtime System for Reactive Programming**. Master thesis, Technische Universität Darmstadt, 2015. JRuby Truffle
- C. Seaton. **Specialising Dynamic Techniques for Implementing the Ruby Programming Langauge**. PhD thesis, University of Manchester, 2015. JRuby Truffle
- M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, H. Mössenböck. **High-Performance Cross-Language Interoperability in a Multi-Language Runtime**. In Proceedings of 11th Dynamic Languages Symposium (DLS), 2015. JRuby Truffle

YARV: Yet Another RubyVM

Innovating the Ruby Interpreter

Koichi Sasada
Graduate School of Technology,
Tokyo University of Agriculture and Technology
2-24-16 Nakacho, Koganei-shi, Tokyo, Japan.
sasada@namikilab.tuat.ac.jp

ABSTRACT

Ruby - an Object-Oriented scripting language - is used world-wide because of its ease of use. However, the current interpreter is slow. To solve this problem, some virtual machines were developed, but none with adequate performance or functionality. To fill this gap, I have developed a Ruby interpreter called *YARV (Yet Another Ruby VM)*. YARV is based on a stack machine architecture and features optimizations for high speed execution of Ruby programs. In this poster, I introduce the Ruby programming language, discuss certain characteristics of Ruby from the aspect of a Ruby interpreter implementer, and explain methods of implementation and optimization. Benchmark results are given at the end.

Keywords

Interpreter Implementation, Scripting Language, Ruby

1. INTRODUCTION

Ruby is the interpreted scripting language developed by Yukihiro Matsumoto for quick and easy object-oriented programming[2, 7]. It is simple, straight-forward, extensible,

- Exception handling
- Closure and method invocation with a block
- Garbage collection support
- Dynamic module loading
- Many useful libraries
- Highly portable

However, current Ruby intepreter is slow. This is because current interpreter (old-ruby) works by traversing abstract syntax tree and evaluates each node. To solve this problem, I have developed new Ruby interpreter called YARV (Yet Another RubyVM), which is a stack machine and runs Ruby programs in compiled intemediate representation of sequential instructions. I'm working to replacing old-ruby with YARV.

This poster is dedicated to discussing the Ruby programming language and the advantages and challenges that Ruby presents as an interpreter target, with speed-up being the

Zero-Overhead Metaprogramming

Reflection and Metaobject Protocols Fast and without Compromises

Stefan Marr
RMoD, Inria, Lille, France
stefan.marr@inria.fr

Chris Seaton
Oracle Labs / University of Manchester
chris.seaton@oracle.com

Stéphane Ducasse
RMoD, Inria, Lille, France
stephane.ducasse@inria.fr

Abstract

Runtime metaprogramming enables many useful applications and is often a convenient solution to solve problems in a generic way, which makes it widely used in frameworks, middleware, and domain-specific languages. However, powerful metaobject protocols are rarely supported and even common concepts such as reflective method invocation or dynamic proxies are not optimized. Solutions proposed in literature either restrict the metaprogramming capabilities or require application or library developers to apply performance improving techniques.

For overhead-free runtime metaprogramming, we demonstrate that *dispatch chains*, a generalized form of polymorphic inline caches common to self-optimizing interpreters, are a simple optimization at the language-implementation level. Our evaluation with self-optimizing interpreters shows that unrestricted metaobject protocols can be realized for the first time without runtime overhead, and that this optimization is applicable for just-in-time compilation of interpreters based on meta-tracing as well as partial evalua-

concise programs. Metaobject protocols (MOPs), as in Smalltalk or CLOS, go beyond more common metaprogramming techniques and enable for example DSL designers to change the language's behavior from within the language [Kiczales et al. 1991]. With these additional capabilities, DLSs can for instance have more restrictive language semantics than the host language they are embedded in.

However, such runtime metaprogramming techniques still exhibit severe overhead on most language implementations. For example, for Java's dynamic proxies we see 6.5x overhead, even so the HotSpot JVM has one of the best just-in-time compilers. Until now, solutions to reduce the runtime overhead either reduce expressiveness of metaprogramming techniques [Masuhara et al. 1995, Chiba 1996, Asai 2014] or burden the application and library developers with applying optimizations [Shali and Cook 2011, DeVito et al. 2014]. One of the most promising solutions so far is trace-based compilation, because it can optimize reflective method invocation or field accesses. However, we found that MOPs still require additional optimizations. Furthermore, trace-based compi-

Dynamic Enforcement of Determinism in a Parallel Scripting Language

Li Lu
University of Rochester
llu@cs.rochester.edu

Weixing Ji
Beijing Institute of Technology
jwx@bit.edu.cn

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

Abstract

Determinism is an appealing property for parallel programs, as it simplifies understanding, reasoning and debugging. It is particularly appealing in dynamic (scripting) languages, where ease of programming is a dominant design goal. Some existing parallel languages use the type system to enforce determinism statically, but this is not generally practical for dynamic languages. In this paper, we describe how determinism can be obtained—and dynamically enforced/verified—for appropriate extensions to a parallel scripting language. Specifically, we introduce the constructs of Deterministic Parallel Ruby (DPR), together with a run-time system (TARDIS) that verifies properties required for determinism, including correct usage of reductions and commutative operators, and the mutual independence (data-race freedom) of concurrent tasks. Experimental results confirm that DPR can provide scalable performance on multicore machines and that the overhead of TARDIS is low enough for practical testing. In particular, TARDIS significantly outperforms alternative data-race detectors with comparable functionality. We conclude with a discussion of future directions in the dynamic enforcement of determinism.

Categories and Subject Descriptors D.1.3 [*Programming Tech-*

We believe determinism to be particularly appealing for dynamic "scripting" languages—Python, Ruby, Javascript, etc. Raw performance has never been a primary goal in these languages, but it seems inevitable that implementations for future many-core machines will need to be successfully multithreaded. At the same time, the emphasis on ease of programming suggests that the tradeoff between simplicity and generality will be biased toward simplicity, where determinism is an ideal fit.

In general, determinism can be obtained either by eliminating races (both data races and synchronization races) in the source program ("language level" determinism), or by ensuring that races are always resolved "the same way" at run time ("system level" determinism). While the latter approach is undeniably useful for debugging, it does not assist with program understanding. We therefore focus on the former.

Language-level determinism rules out many historically important programming idioms, but it leaves many others intact, and it offers significant conceptual benefits, eliminating many of the most pernicious kinds of bugs, and allowing human readers to more easily predict a program's behavior from its source code. Using well-known constructs from previous parallel languages, we have defined a language dialect we call Deterministic Parallel Ruby

Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira
IBM Research – Tokyo
odaira@jp.ibm.com

Jose G. Castanos
IBM Research – T.J. Watson Research Center
castanos@us.ibm.com

Hisanobu Tomari
University of Tokyo
tomari@is.s.u-tokyo.ac.jp

Abstract

Many scripting languages use a Global Interpreter Lock (GIL) to simplify the internal designs of their interpreters, but this kind of lock severely lowers the multi-thread performance on multi-core machines. This paper presents our first results eliminating the GIL in Ruby using Hardware Transactional Memory (HTM) in the IBM zEnterprise EC12 and Intel 4th Generation Core processors. Though prior prototypes replaced a GIL with HTM, we tested realistic programs, the Ruby NAS Parallel Benchmarks (NPB), the WEBrick HTTP server, and Ruby on Rails. We devised a new technique to dynamically adjust the transaction lengths on a per-bytecode basis, so that we can optimize the

The single-thread performance is limited because of interpreted execution, dynamic typing, and the support for metaprogramming. Many projects [10,24,30] have attempted to overcome these limitations through Just-in-Time (JIT) compilation with type specialization.

Meanwhile, the multi-thread performance of the scripting languages is constrained by the Global Interpreter Lock (GIL), or the Giant VM Lock (GVL) in Ruby's terminology. Although each application thread is mapped to a native thread, only the one thread that acquires the GIL can actually run. At pre-defined yield points, each thread releases the GIL, yields the CPU to another runnable thread if any exists, and then reacquires the GIL. The GIL eases the pro-

Static Type Inference for Ruby

Michael Furr Jong-hoon (David) An Jeffrey S. Foster Michael Hicks

Department of Computer Science
University of Maryland
College Park, MD 20742
{furr,davidan,jfoster,mwh}@cs.umd.edu

ABSTRACT

Many general-purpose, object-oriented scripting languages are dynamically typed, which provides flexibility but leaves the programmer without the benefits of static typing, including early error detection and the documentation provided by type annotations. This paper describes Diamondback Ruby (DRuby), a tool that blends Ruby's dynamic type system with a static typing discipline. DRuby provides a type language that is rich enough to precisely type Ruby code we have encountered, without unneeded complexity. When possible, DRuby infers static types to discover type errors in Ruby programs. When necessary, the programmer can provide DRuby with annotations that assign static types to dynamic code. These annotations are checked at run time, isolating type errors to unverified code. We applied DRuby to a suite of benchmarks and found several bugs that would cause run-time type errors. DRuby also reported a number of warnings that reveal questionable programming

However, this flexibility comes at a price. Programming mistakes that would be caught by static typing, e.g., calling a method with the wrong argument types, remain latent until run time. Such errors can be painful to track down, especially in larger programs. Moreover, with pure dynamic typing, programmers lose the concise, automatically-checked documentation provided by type annotations. For example, the Ruby standard library includes textual descriptions of types, but they are not used by the Ruby interpreter in any way, and in fact, we found several mistakes in these ad hoc type signatures in the process of performing this research.

To address this situation, we have developed Diamondback Ruby (DRuby), an extension to Ruby that blends the benefits of static and dynamic typing. Our aim is to add a typing discipline that is simple for programmers to use, flexible enough to handle common idioms, that provides programmers with additional checking where they want it, and reverts to run-time checks where necessary. DRuby is focused on Ruby, but we expect the advances we make

SMALLTALK-80

THE LANGUAGE AND ITS IMPLEMENTATION

(CALL)

$$\frac{\begin{array}{c}\mathcal{H};\mathcal{E};e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';v \mid \phi' \\ \mathcal{H}';\mathcal{E}';e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';l \mid \phi'' \\ \mathcal{H}''(l) = A\langle\_\rangle \qquad \mathcal{C}(A) = \mathcal{M} \qquad \mathcal{M}(m) = \lambda(x)e'' \\ \overline{\phi} \in \mathcal{C} \qquad \mathcal{H}'';\{\mathsf{self} \mapsto l, x \mapsto v\};e'' \mid \overline{\phi} \longrightarrow_{\mathcal{C}} \mathcal{H}''';\_;v' \mid \_\end{array}}{\mathcal{H};\mathcal{E};e'.m(e) \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}''';\mathcal{E}'';v' \mid \phi''}$$

(MCOND)

$$\frac{\begin{array}{c}\mathcal{H};\mathcal{E};e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';v \mid p,\phi' \\ p = \ell, e_p = e' \text{ if } v \neq \mathsf{nil} \qquad p = \neg\ell, e_p = e'' \text{ if } v = \mathsf{nil} \\ \mathcal{H}';\mathcal{E}';e_p \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';v' \mid \phi''\end{array}}{\mathcal{H};\mathcal{E};\mathsf{if}^{\ell}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';v' \mid \phi''}$$

(MCALL)

$$\frac{\begin{array}{c}\mathcal{H};\mathcal{E};e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';v \mid \phi' \\ \mathcal{H}';\mathcal{E}';e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';l \mid \phi'' \\ \mathcal{H}''(l) = A\langle\_\rangle \qquad \mathcal{C}(A) = \mathcal{M} \\ \mathcal{M}(m) = \lambda(x)e'' \qquad \mathtt{runtype}_{\mathcal{H}''}(v) \leq \mathcal{C}(A.m) \\ \overline{\phi} \in \mathcal{C} \qquad \mathcal{H}'';\{\mathsf{self} \mapsto l, x \mapsto v\};e'' \mid \overline{\phi} \longrightarrow_{\mathcal{C}} \mathcal{H}''';\_;v'' \mid \_\end{array}}{\mathcal{H};\mathcal{E};\underline{e'.m(e)} \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}''';\mathcal{E}'';v'' \mid \phi''}$$

Chris Seaton

Oracle Labs / University of Manchester

chris.seaton@oracle.com

rubybib.org