P G22.3033-002
Scripting Languages: Performance of Dynamic Scripting Languages (main focus: Python)

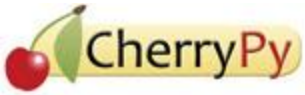
Priya Nagpurkar, IBM Research

August 2, 2012

Popularity of Scripting Languages

- High productivity, quick and simple prototyping
  - Language features like dynamic typing, high level data structures
  - Frameworks that simplify development and deployment: Rails (Ruby), Django and Zope (Python)
- Popular even in emerging server application domains
  - Cloud: Google AppEngine
  - Web 2.0: FaceBook (PHP/Python), YouTube (Python), Twitter (Ruby)
- Python application domains:

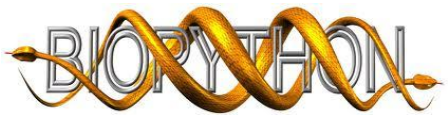
Web applications, frameworks, servers



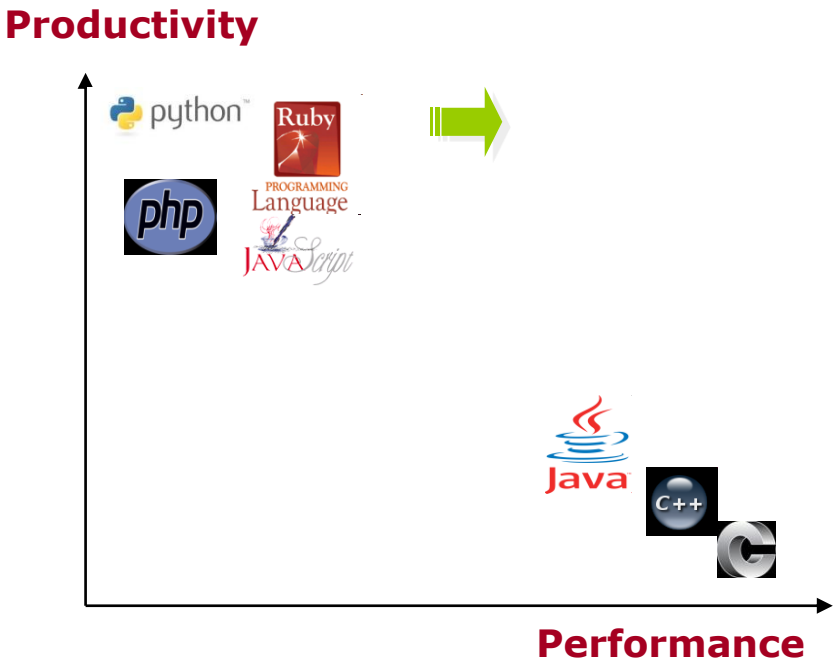
Enterprise applications

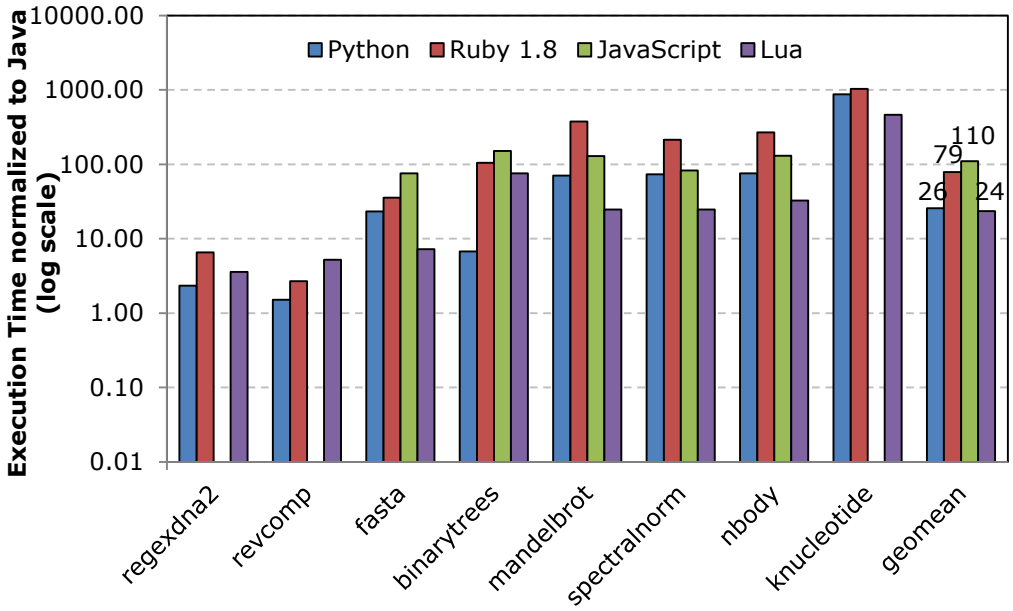
Systems management

Scientific Computing



The Performance Gap

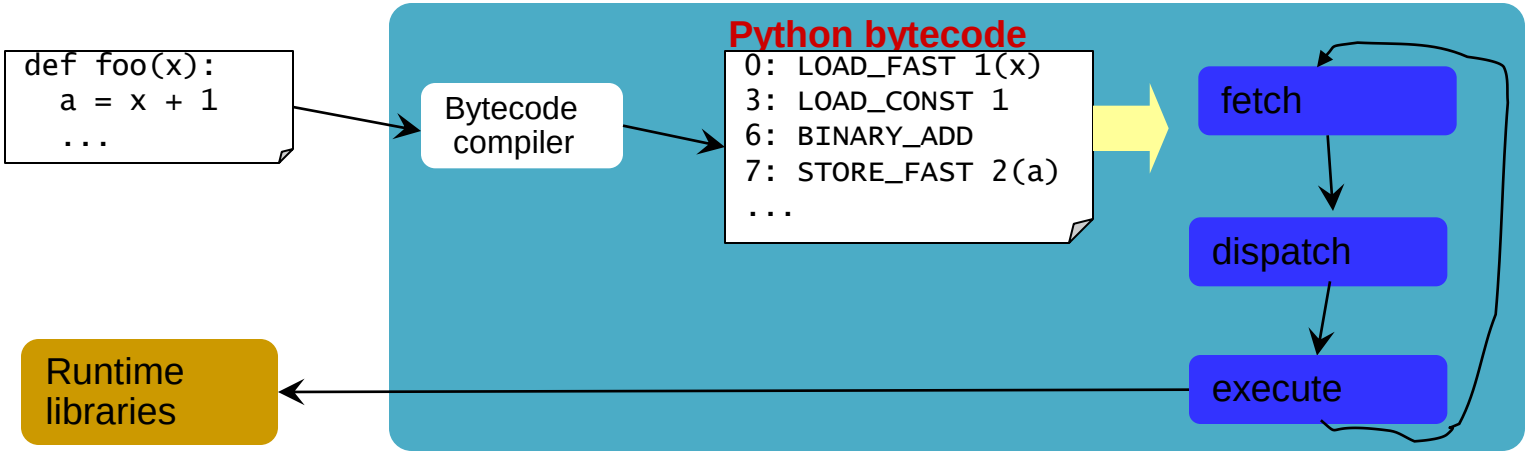




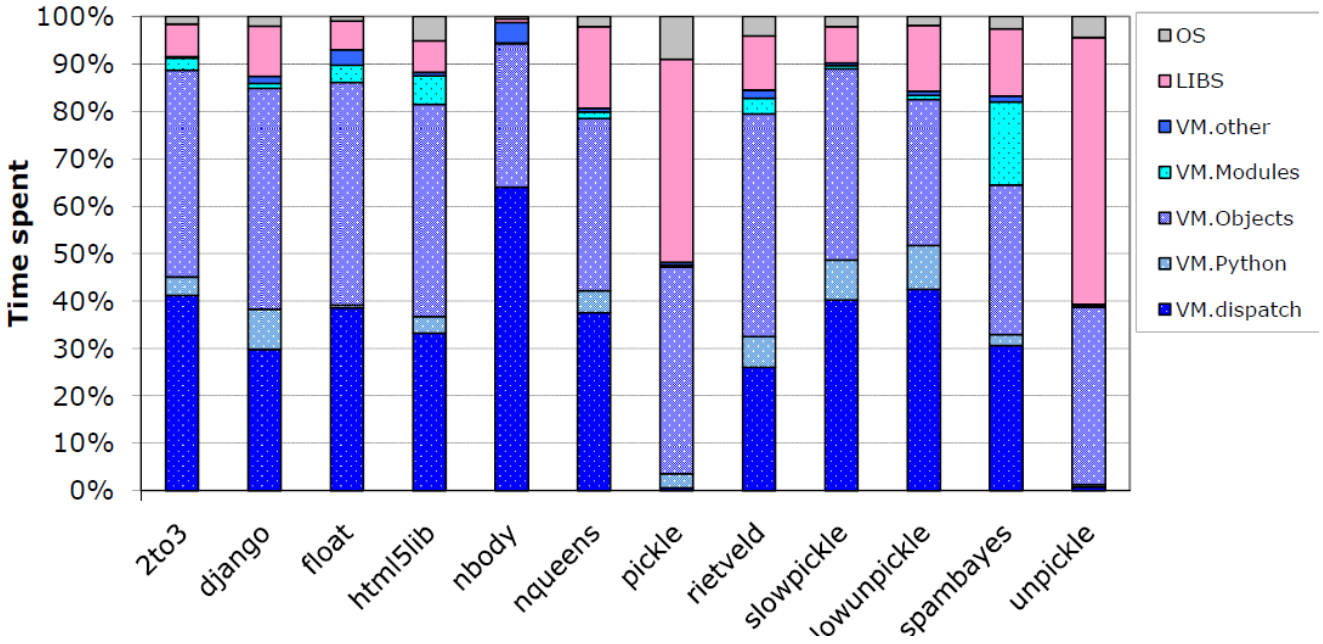
- Focus primarily on ease of programming, flexibility, and features, not on performance
- Original, and often the most commonly used implementations are interpreters
  - Easy to add new features, portable, lightweight
  - Also inefficient and very slow!

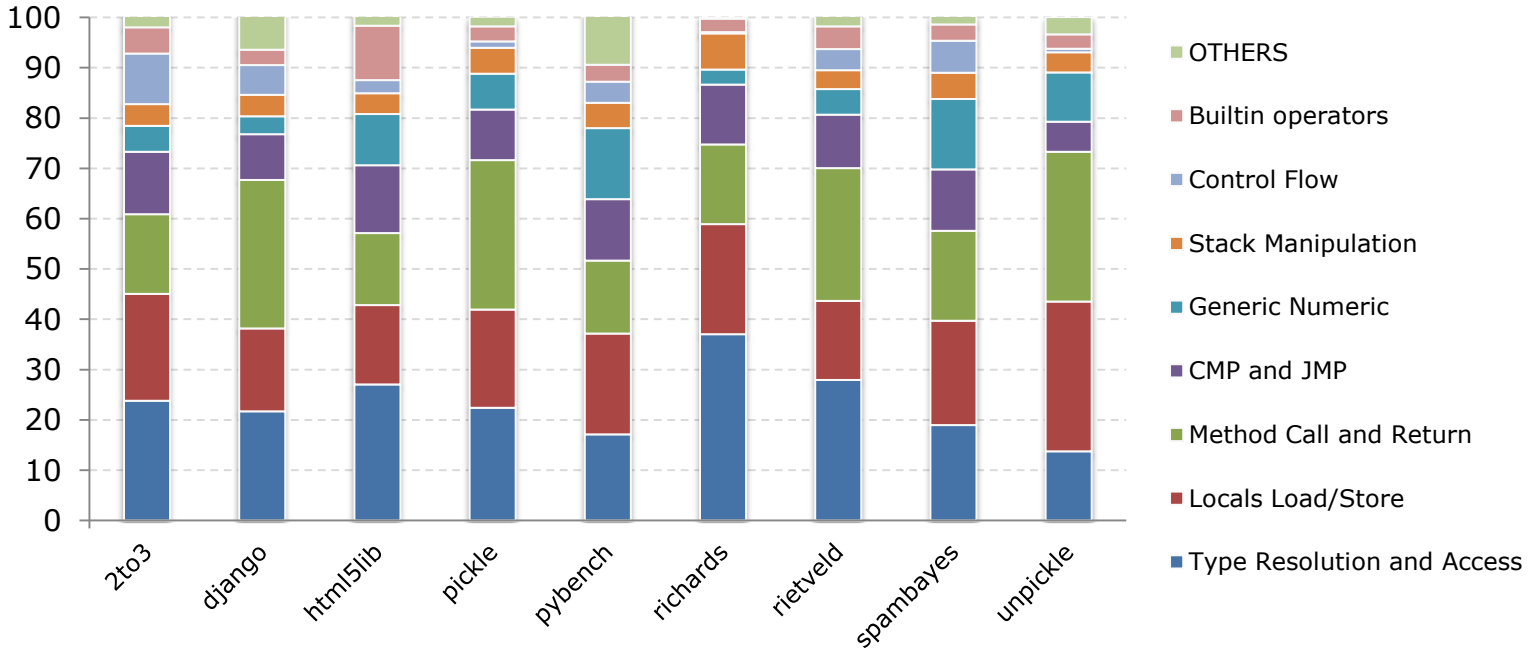
Dynamic Scripting Languages: Implementation Highlights

- Popular and widely used implementations are interpreters written in C
  - Cpython 2.x, Ruby 1.9
- Interpretation and dynamic typing introduce lots of:
  - Indirect branches
  - Indirect memory accesses
  - Indirect function calls
- Fat bytecodes make interpreter dispatch (big switch statement) less significant as compared to interpreted Java, threading improves dispatch
  - 36x on Java vs. 2x on Python, for n-body benchmark
- Frequent use of shared libraries

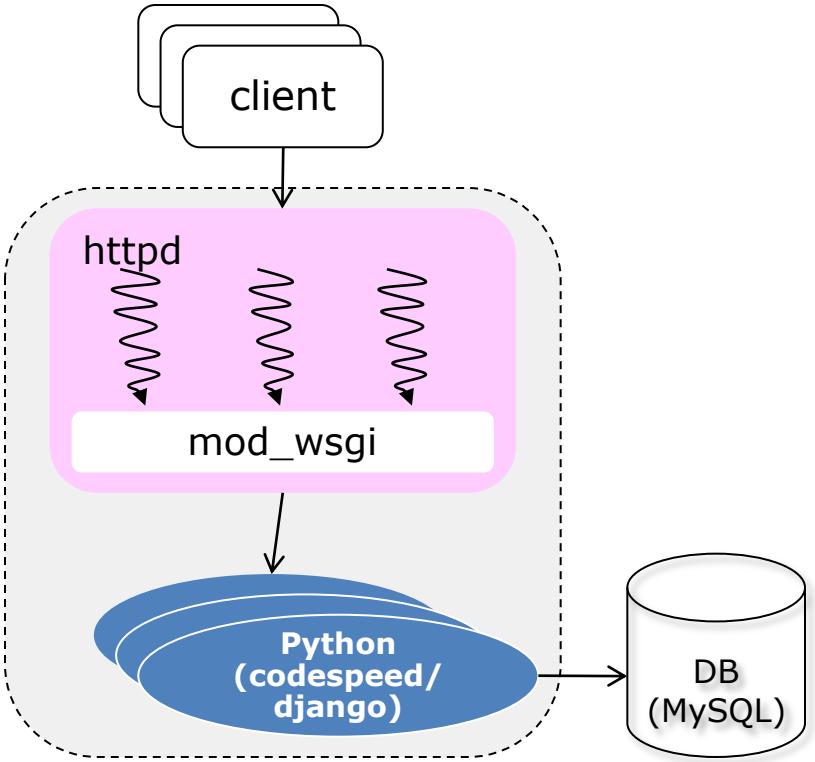


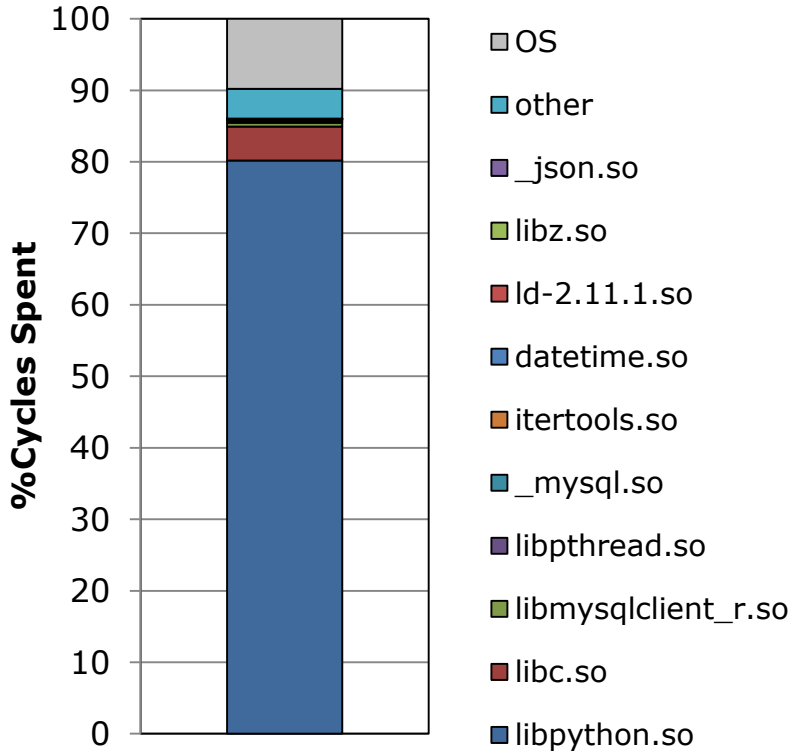
Where is the time spent? (Unladen-Swallow benchmarks)





Where is the time spent? (Python Web Application)





- Codespeed: Python web application used to track performance over software revisions for multiple projects (used by the PyPy, Twisted projects)
- Three-tier application that uses the python django web-application framework and a backend database, deployed using apache httpd
- Request-level parallelism is achieved using multiple httpd processes/threads, each with an embedded python interpreter instance

At which level should we optimize?

- Improve interpreter performance through runtime techniques
- Add JIT compilation
  - Method based vs. trace based compilation
  - Speculative optimization
- Perform dynamic binary optimization of interpreted code
- Evaluate and optimize microarchitecture for interpreters
  - How to run poorly optimized code well
- Provide architectural support for:
  - Speculative optimization
  - Frequent and expensive operations: e.g. type checking

  (Checked Load: Architectural support for JavaScript type-checking on mobile processors, O. Anderson, E. Fortuna, L. Ceze, S. Eggers, HPCA 2011)

Some existing approaches

Improving existing interpreters
- Zend: PHP

New Interpreter and JIT
- LuaJIT
- PyPy

Mapping to existing managed runtime
- On top of CLR (.NET)
- Jython/JRuby on Java Virtual Machine (JVM)
- Oracle (BEA): PHP on JVM

Extending existing managed runtime for Dynamic Scripting Languages
- Microsoft DLR: IronPython, IronRuby, IronJscript
- Microsoft Mono: open source CLR implementation
- Sun DaVinci (JVM): multi-language

Add JIT to existing interpreters
- Unladen-swallow (Google) : CPython + open source LLVM JIT
- Rubinius: Ruby + open source LLVM JIT
- Psyco: CPython + mini-JIT

Convert to static language (C/C++)
- Cython/Pyrex: Python
- HipHop (FaceBook): PHP

Improving interpreter performance: example

- Global variable and Attribute Caching
  - Addresses the overhead associated with loading global variables and object attributes (fields and methods)
  - Global variables and object attributes are stored in dictionaries and looked up by name
    - A single read can involve complex control flow, indirect function calls, and multiple dictionary lookups
  - Global variable caching caches the address of the dictionary entry for a global variable along with a version number
    - Change in value does not invalidate cached address
    - Change in shape of dictionary is tracked using version numbers
  - Global variable caching and attribute caching together lead to a 7% improvement in performance (on an average across the unladen-swallow benchmark suite)

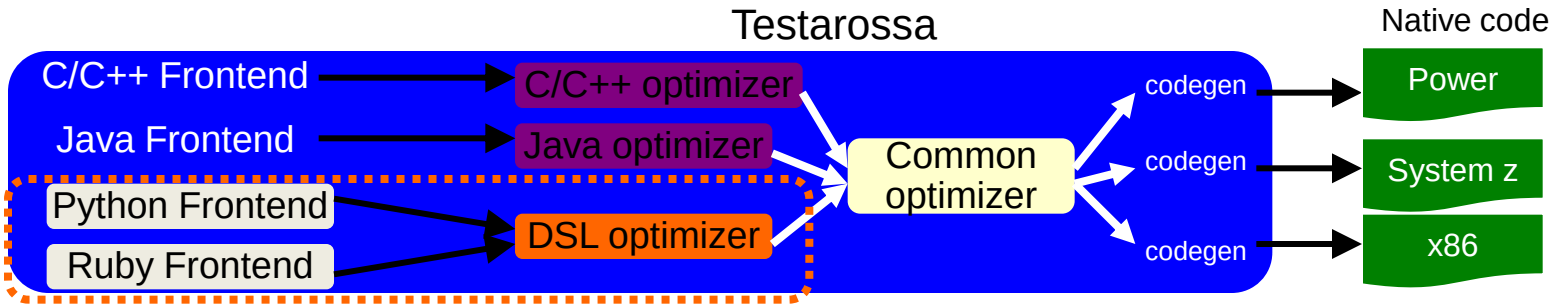
More details in UCSB tech report: Understanding the Potential of Interpreter-based Optimizations for Python

Improving Performance through Compilation: Challenges

- Dynamic typing and metaprogramming make static analysis and optimization difficult
- Large and potentially changing variety of languages makes per-language approaches non-scalable
- Moving target:
  - Many of these languages continue to evolve
  - Deployment practices and popular components (web servers, frameworks) are also evolving
- Compatibility with significant amount of existing code in the form of reusable modules/libraries and applications important

Adaptive Just in Time Compilation

- IBM Research Project: Fiorano JIT Compiler for Dynamic Scripting Languages
- Enable Dynamic scripting language support on top of IBM's Testarossa compiler (TR)
  - Leverage TR as the optimization framework and backend for native code generation
  - Handle multiple languages with minimal engineering effort by creating a generic Dynamic Scripting Languages optimization framework
  - Attach Fiorano JIT compiler to open source language virtual machines
    - Preserve original language semantics and compatibility to deal with rapid language evolution
- Current performance: 63% improvement over CPython



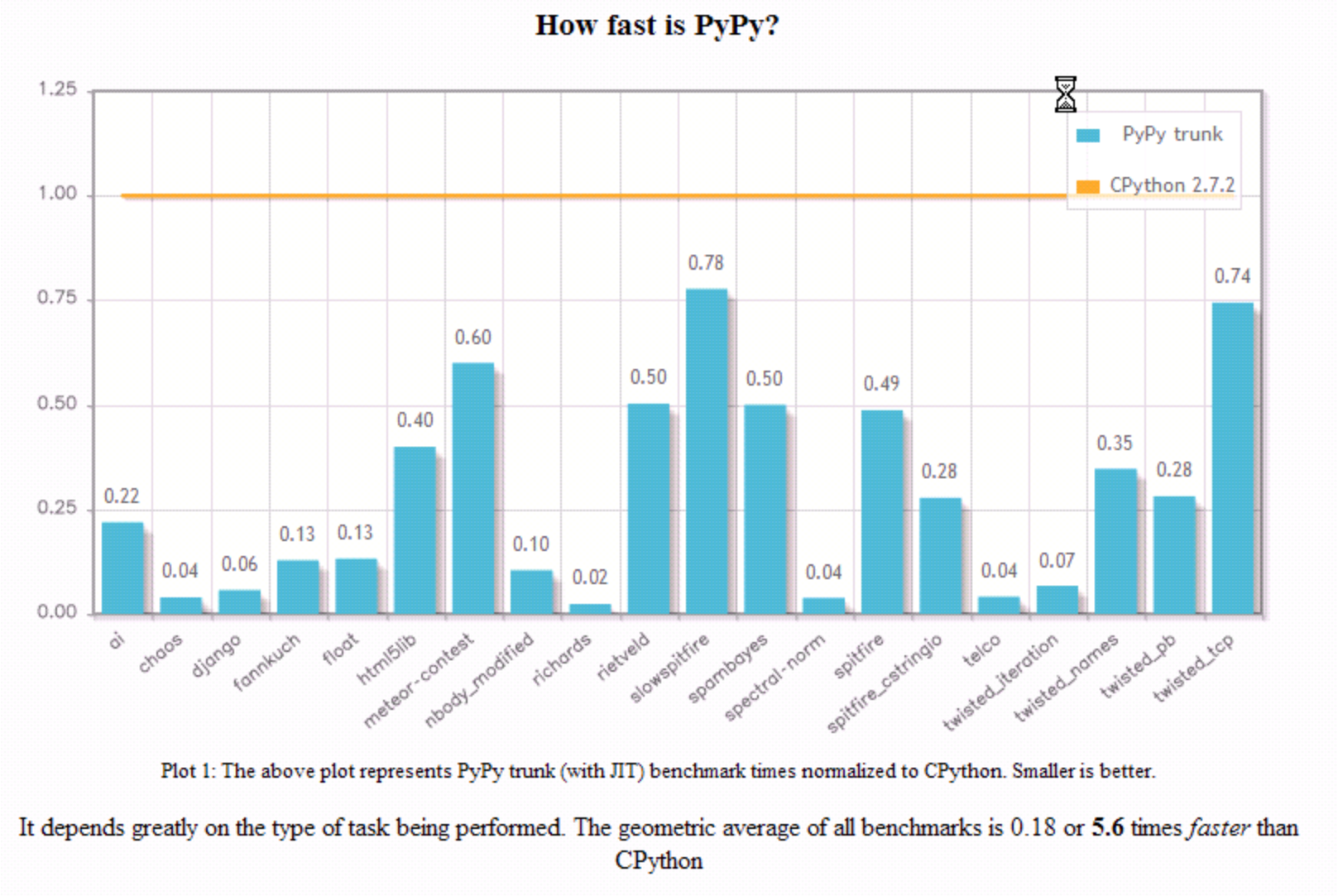
Improving Performance through Compilation: Effective techniques

- Hidden Classes (Self, v8)
  - Optimize class offsets and accesses
- Tracing (TraceMonkey, LuaJIT, PyPy)
  - Method + Tracing (JaegerMonkey)
- Profiling Feedback (Unladen Swallow, PyPy)
  - Type
  - Value
  - Global state (dictionary, …)
- Unboxing
- DynamicSites (DLR), InvokeDynamic (Java7/DaVinci)
  - Cache method lookup, specialized implementation
  - Polymorphic Inline Cache
  - Type inference and propagation

PyPy

- A Python implementation written in Rpython
  - Implementations for other languages also exist
- RPython is a restricted version of Python
  - *Well-typed* according to type inference rules of RPython
  - Class definitions do not change
  - Object model implementation exposes runtime constants
  - Various hints to trace selection engine to capture user program scope
- Tracing JIT through both user program and runtime
  - A trace is a single-entry-multiple-exit code sequence (like long extended basic block)
  - Tracing automatically incorporates runtime feedback and guards into the trace
- The optimizer fully exploits the simple topology of a trace to do very powerful data-flow based redundancy elimination

PyPy Performance

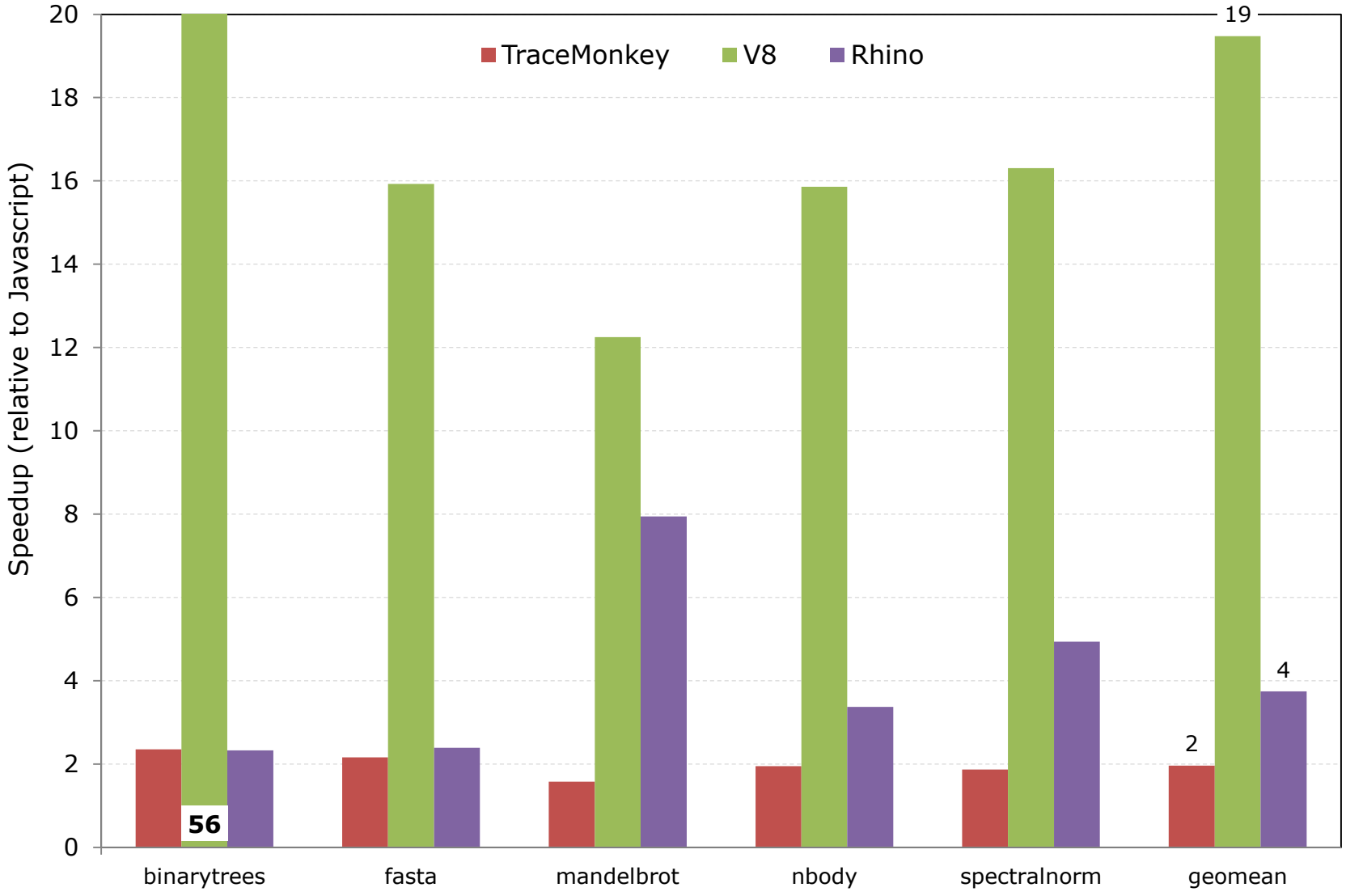


Plot 1: The above plot represents PyPy trunk (with JIT) benchmark times normalized to CPython. Smaller is better.

It depends greatly on the type of task being performed. The geometric average of all benchmarks is 0.18 or **5.6** times *faster* than CPython

- Source: http://speed.pypy.org/

Performance of JavaScript Implementations



Thank You!

Questions?