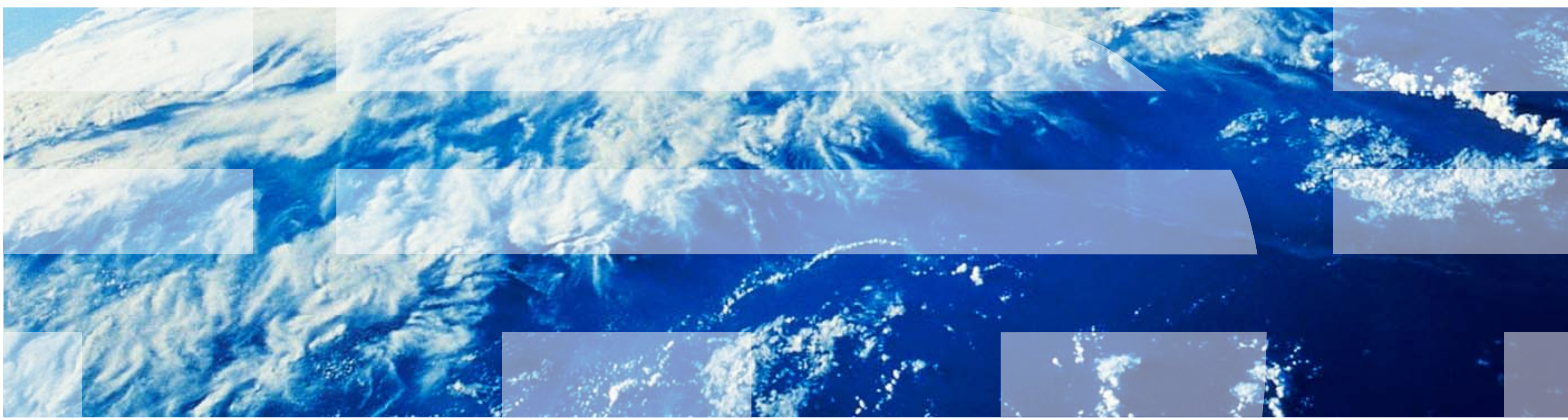
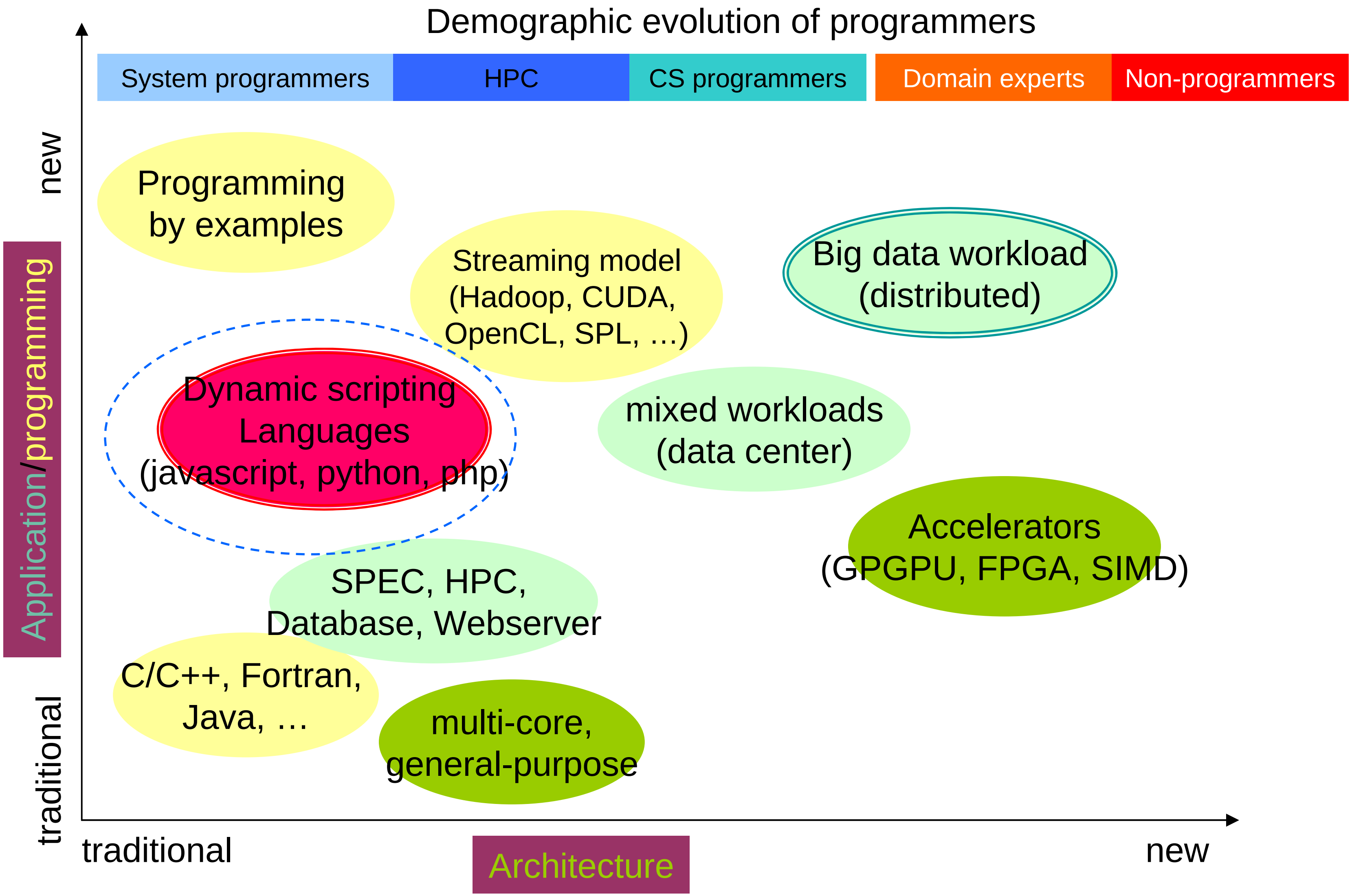
Reusing JITs are from Mars, Dynamic Scripting Languages are from Venus

Peng Wu, IBM T.J. Watson Research Center



Trends in Workloads, Languages, and Architectures



Popularity of Dynamic Scripting Languages

- Trend in emerging programming paradigms
  - **Dynamic scripting languages** are gaining popularity and emerging in production deployment

Commercial deployment
- PHP: Facebook, LAMP
- Python: YouTube, InviteMedia, Google AppEngine
- Ruby on Rails: Twitter, ManyEyes

Education
- Increasing adoption of Python as entry-level programming language

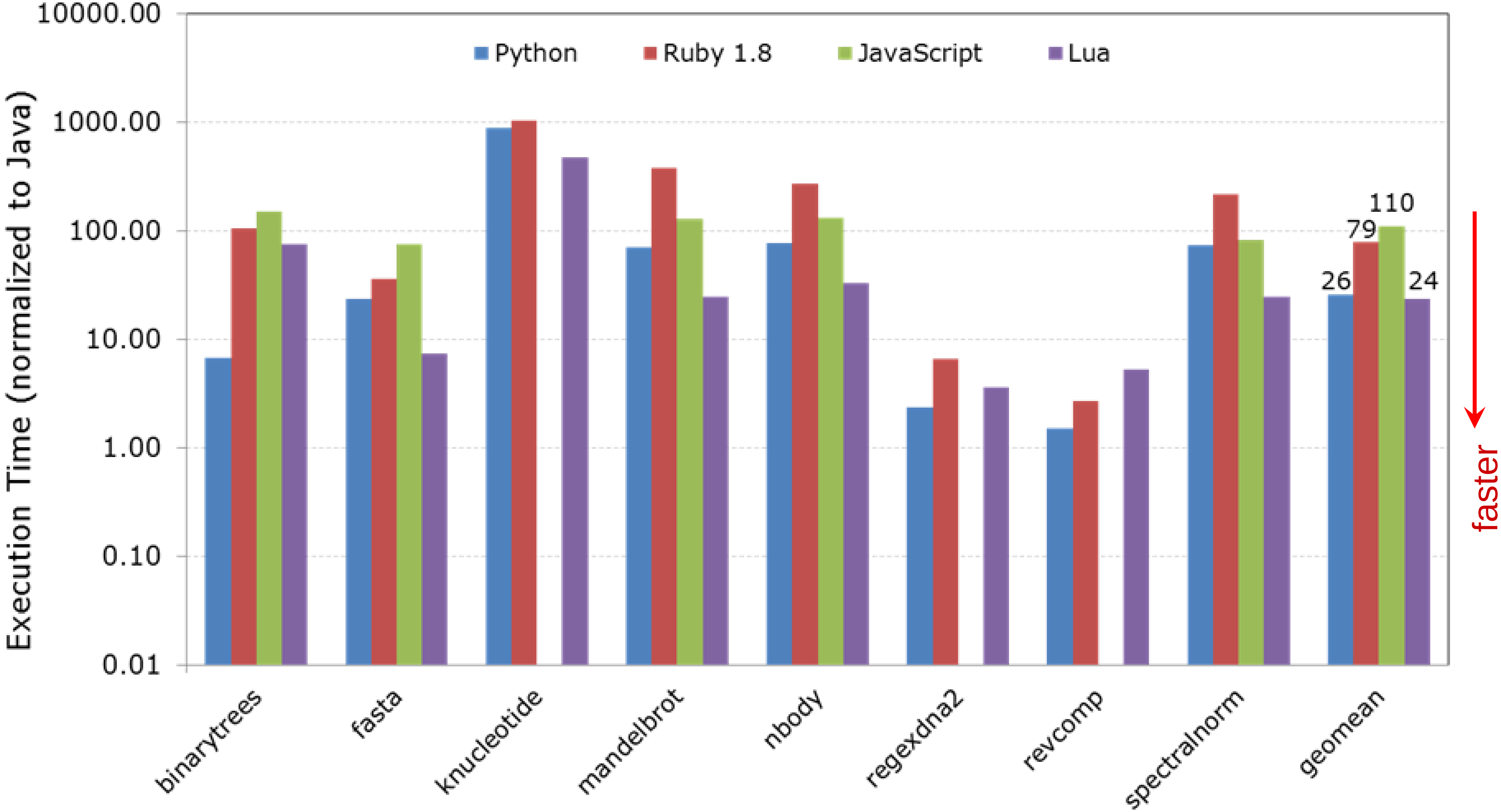
Demographics
- Programming becomes a everyday skill for many non-CS majors

> "Python helped us gain a huge lead in features and a majority of early market share over our competition using C and Java."
>
> \- Scott Becker
>
> CTO of Invite Media Built on Django, Zenoss, Zope

TIOBE Language Index

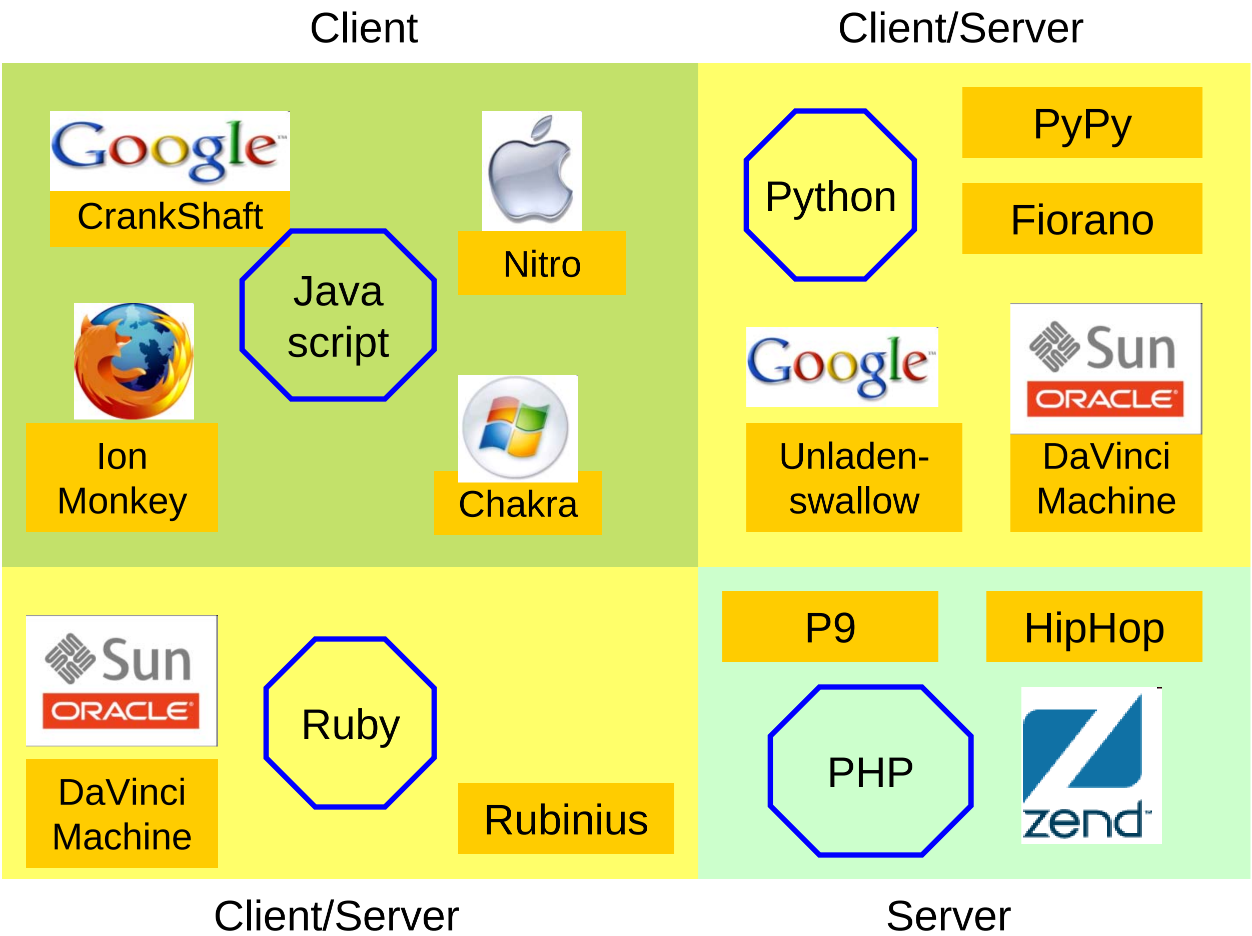
Rank	Name	Share
1	C	17.555%
2	Java	17.026%
3	C++	8.896%
4	Objective-C	8.236%
5	C#	7.348%
6	PHP	5.288%
7	Visual Basic	4.962%
8	Python	3.665%
9	Javascript	2.879%
10	Perl	2.387%
11	Ruby	1.510%

Language Interpreter Comparison (Shootout)



Benchmarks: shootout (http://shootout.alioth.debian.org/) measured on Nehalem
Languages: Java (JIT, steady-version); Python, Ruby, Javascript, Lua (Interpreter)
Standard DSL implementation (interpreted) can be 10~100 slower than Java (JIT)

Dynamic Scripting Language JIT Landscape



Significant difference in JIT effectiveness across languages
- Javascript has the most effective JITs
- Ruby JITs are similar to Python's

❑JVM based
- Jython
- JRuby
- Rhino

❑CLR based
- IronPython
- IronRuby
- IronJscript
- SPUR

❑Add-on JIT
- Unladen-swallow
- Fiorano
- Rubinius

❑Add-on trace JIT
- PyPy
- LuaJIT
- TraceMonkey
- SPUR

Scripting Languages Compilers: A Tale of Two Worlds

- Customary VM and JIT design targeting one scripting language
  - in-house VM developed from scratch and designed to facilitate the JIT
  - in-house JIT that understands target language semantics
- Heavy development investment, most noticeably in Javascript
  - where performance transfers to competitiveness
- Such VM+JIT bundle significantly reduces the performance gap between scripting languages and statically typed ones
  - Sometimes more than 10x speedups over interpreters

- The reusing JIT phenomenon
  - reuse the prevalent interpreter implementation of a scripting language
  - attach an existing mature JIT
  - (optionally) extend the "reusing" JIT to optimize target scripting languages
- Considerations for reusing JITs
  - Reuse common services from mature JIT infrastructure
  - Harvest the benefits of mature optimizations
  - Compatibility with standard implementation by reusing VM
- Willing to sacrifice some performance, but still expect substantial speedups from compilation

Scripting Languages Compilers: A Tale of Two Worlds

- Customary VM and JIT design targeting one scripting language
  - in-house VM developed from scratch and designed to facilitate the JIT
  - in-house JIT that understands target language semantics
- Heavy development investment, most noticeably in Javascript
  - where performance transfers to competitiveness
- Such VM+JIT bundle significantly reduces the performance gap between scripting languages and statically typed ones
  - Sometimes more than 10x speedups over interpreters

- The reusing JIT phenomenon
  - reuse the prevalent interpreter implementation of a scripting language
  - attach an existing mature JIT
  - (optionally) extend the “reusing” JIT to optimize target scripting languages
- Considerations for reusing JITs
  - Reuse common services from mature JIT infrastructure
  - Harvest the benefits of mature optimizations
  - Compatibility with standard implementation by reusing VM
- Willing to sacrifice some performance, but still expect substantial speedups from compilation

Outline

Let’s take an in-depth look at the reusing JIT phenomenon

We focus on the world of Python JIT

1. PyPy: customary VM + trace JIT based on RPython
2. Fiorano JIT: based on Testarossa JIT from IBM J9 VM (our own)
3. Jython: translating Python codes into Java codes
4. Unladen-swallow JIT: based on LLVM JIT (google)
5. IronPython: translating Python codes into CLR (Microsoft)

The rest of the talk

– The state-of-the-art of reusing JIT approach
– Understanding Jython, Fiorano JIT, and PyPy
– Recommendation of Reusing JIT designers
– Conclusions

Python Language and Implementation

- Python is an object-oriented, dynamically typed language
  - Monolithic object model (every data is an object, including integer or method frame)
  - support exception, garbage collection, function continuation
  - CPython is Python interpreter in C (de factor standard implementation of Python)

foo.py

```
def foo(list):
     return len(list)+1
```

python bytecode

```
 0 LOAD_GLOBAL      0 (len)
 3 LOAD_FAST        0 (list)
 6 CALL_FUNCTION    1
 9 LOAD_CONST       1 (1)
12 BINARY_ADD
13 RETURN_VALUE
```

- LOAD_GLOBAL (name resolution)
  - dictionary lookup
- CALL_FUNCTION (method invocation)
  - frame object, argument list processing, dispatch according to types of calls
- BINARY_ADD (type generic operation)
  - dispatch according to types, object creation

Overview on Jython

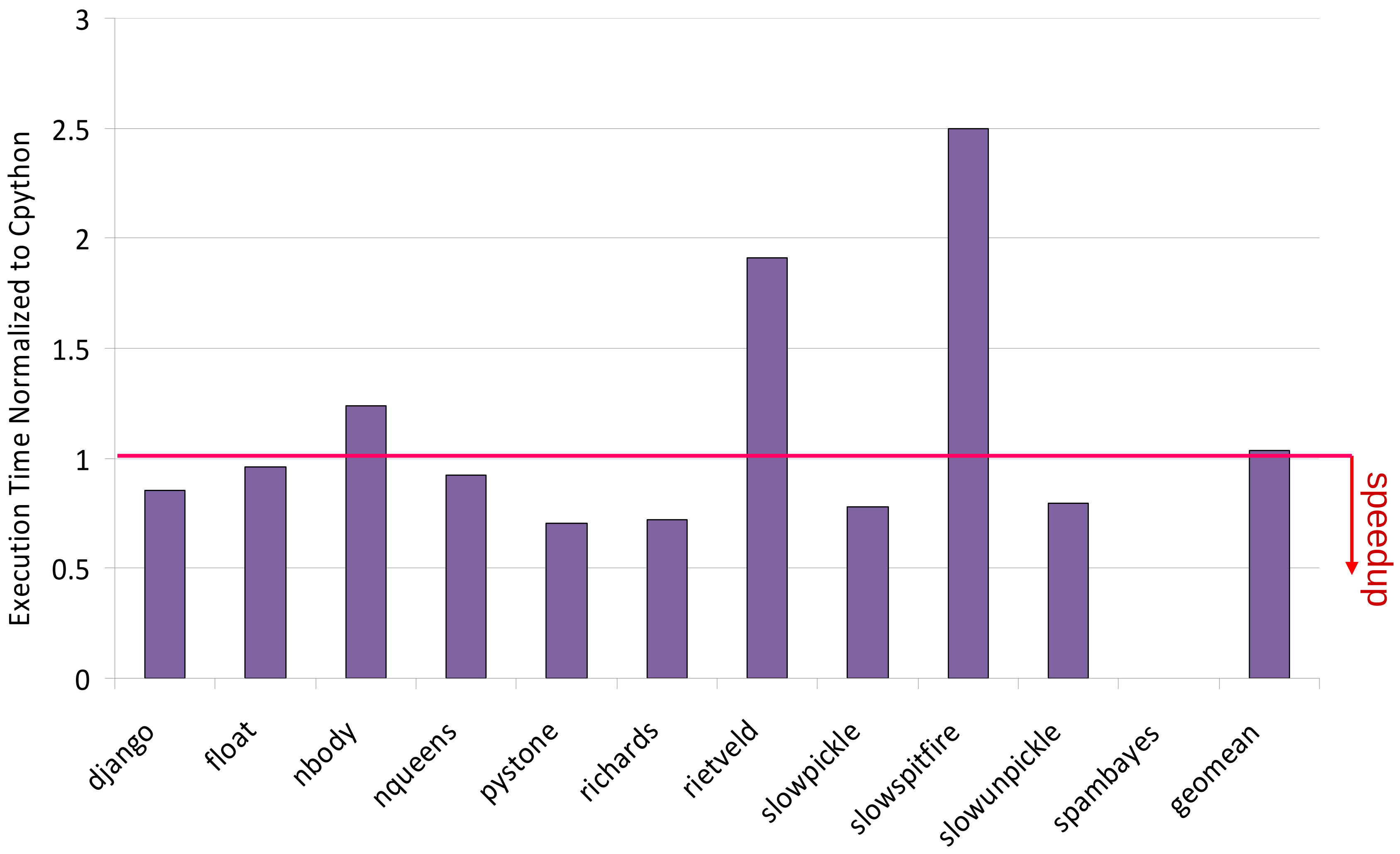
- A clean implementation of Python on top of JVM

- Generate JVM bytecodes from Python 2.5 codes
  - interface with Java programs
  - true concurrence (i.e., no global interpreter lock)
  - but cannot easily support standard C modules

- Runtime rewritten in Java, JIT optimizes user programs and runtime
  - Python built-in objects are mapped to Java class hierarchy
  - Jython 2.5.x does not use InvokeDynamic in Java7 specification

- Jython is an example of JVM languages that share similar characteristics
  - e.g., JRuby, Clojure, Scala, Rhino, Groovy, etc
  - similar to CLR/.NET based language such as IronPython, IronRuby

Execution Time of Jython 2.5.2 Normalized over CPython



Jython: An Extreme case of Reusing JITs

Jython has minimal customization for the target language Python

- It does a “vanilla” translation of a Python program to a Java program
- The (Java) JIT has no knowledge of Python language nor its runtime

```
def calc1(self,res,size):
    x = 0
    while x < size:
        res += 1
        x += 1
    return res
```

```
private static PyObject calc$1(PyFrame frame)
{
  frame.setlocal(3, i$0);
  frame.setlocal(2, i$0);
  while(frame.getlocal(3)._lt(frame.getlocal(0)).__nonzero__())
  {
    frame.setlocal(2, frame.getlocal(2)._add(frame.getlocal(1)));
    frame.setlocal(3, frame.getlocal(3)._add(i$1));
  }
  return frame.getlocal(2);
}
```

Jython Runtime Profile

```
def calc1(self,res,size):
    x = 0
    while x < size:
        res += 1
        x += 1
    return res
```

(a) localvar-loop

```
def calc2(self,res,size):
    x = 0
    while x < size:
        res += self.a
        x += 1
    return res
```

(b) getattr-loop

```
def foo(self):
    return 1

def calc3(self,res,size):
    x = 0
    while x < size:
        res += self.foo()
        x += 1
    return res
```

(c) call-loop

# Java bytecode	path length per Python loop iteration		
	(a) localvar-loop	(b) getattr-loop	(c) call-loop
heap-read	47	80	131
heap-write	11	11	31
heap-alloc	2	2	5
branch	46	70	101
invoke (JNI)	**70(2)**	**92(2)**	**115(4)**
return	70	92	115
arithmetic	18	56	67
local/const	268	427	583
Total	**534**	**832**	**1152**

In an ideal code generation

Critical path of 1 iteration include:

- 2 integer add
- 1 integer compare
- 1 conditional branch

On the loop exit
- box the accumulated value into `PyInteger`
- store boxed value to `res`

100x path length explosion

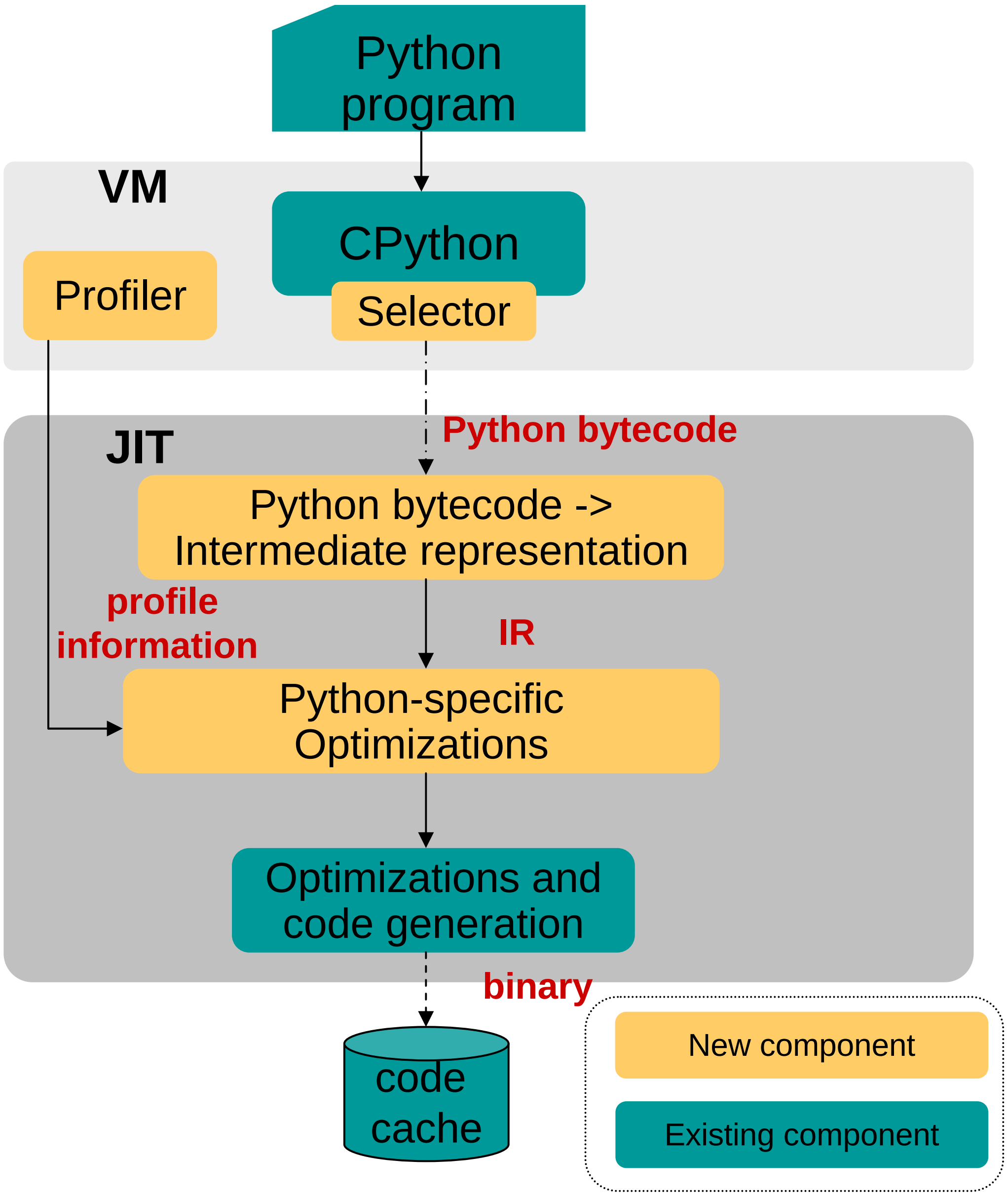
Why is the Java JIT Ineffective?

What does it take to optimize this example effectively?

- ❑ Massive inlining to expose all computation within the loop to the JIT
  - for integer reduction loop, 70 ~ 110 call sites need to be inlined
- ❑ Precise data-flow information in the face of many data-flow join
  - for integer reduction loop, between 40 ~ 100 branches
- ❑ Ability to remove redundant allocation, heap-read, and heap-write
  - require precise alias/points-to information
- ❑ Let's assume that the optimizer can handle local accesses effectively

The Fiorano JIT

- ❑IBM production-quality Just-In-Time (JIT) compiler for Java as a base
- ❑CPython as a language virtual machine (VM)
  - – de facto standard of Python
- ❑Same structure as Unladen Swallow
  - ❑CPython with LLVM

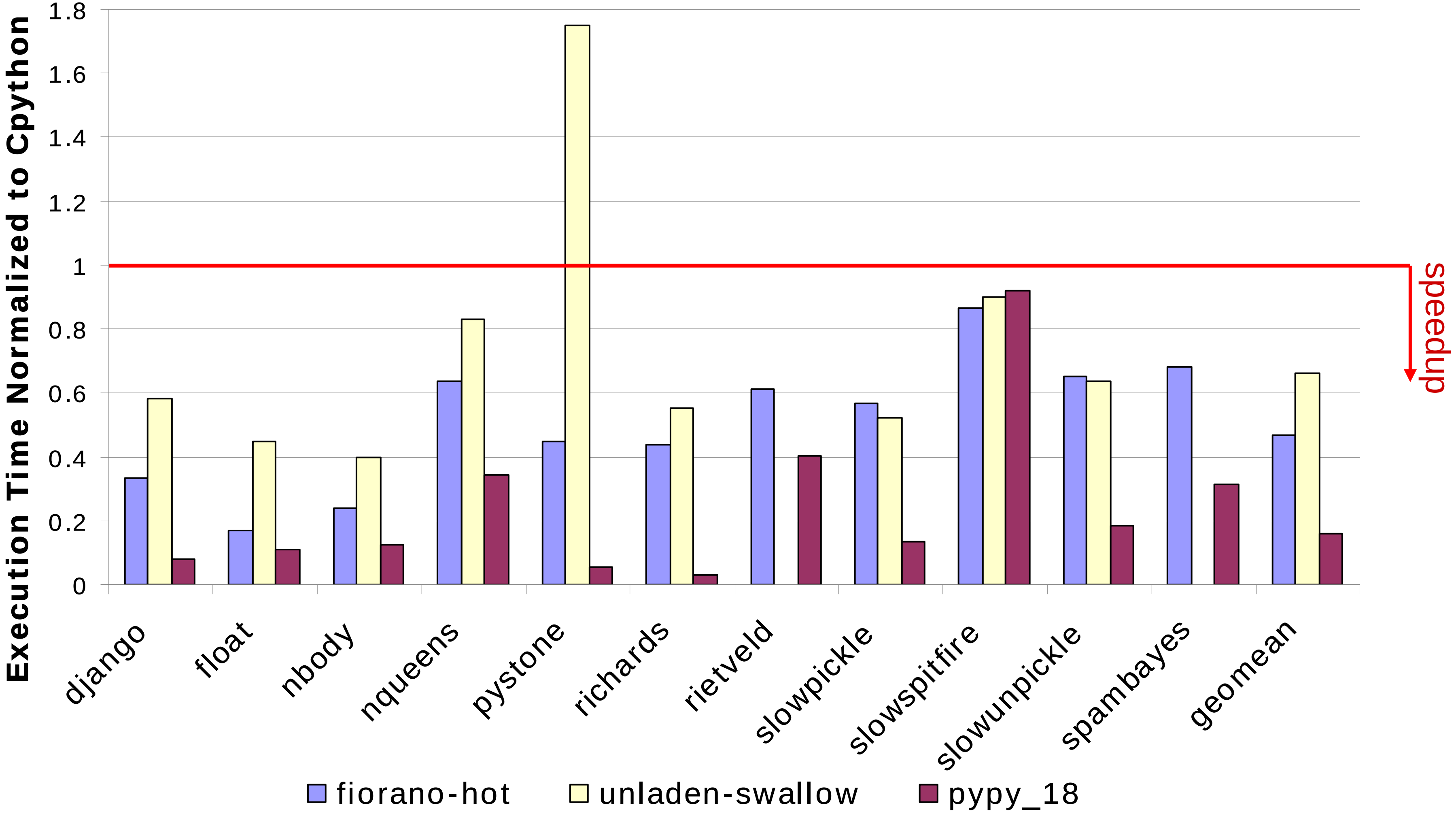


What's Added to the Fiorano JIT?

- No-opt level compilation support
  - Translated CPython bytecode into Testaross IR (IRGEN)
  - Added method hotness profiling and compilation trigger
- Python-specific optimization support
  - Runtime profiling in CPython interpreter
  - A lot of IRGEN level specialization for Python
    - Caching the results of LOAD_GLOBAL (watch invalidation)
    - Fast path versioning for LOAD_ATTR/STORE_ATTR/CALL
    - Guard-based specialization for arithmetic & compare
    - Specialization for built-ins such as instanceof, xrange, sin, cos
    - Guard-based & fast path versioning for GET_ITER/FOR_ITER,UNPACK_SEQUENCE
  - Unboxing optimization for some integer and float
    - Extending the escape analysis optimization in the Testarossa JIT

VEE 2011: Adding Dynamically-Typed Language Support to a Statically-Typed Language Compiler: Performance Evaluation, Analysis, and Tradeoffs

Normalized Execution Time of Python JITs over CPython



PyPy (Customary Interpreter + JIT)

- A Python implementation written in RPython
  - – interface with CPython modules may take a big performance hit

- RPython is a restricted version of Python, e.g., (after start-up time)
  - – *Well-typed* according to type inference rules of RPython
  - – Class definitions do not change
  - – Tuple, list, dictionary are homogeneous (across elements)
  - – Object model implementation exposes runtime constants
  - – Various hint 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 exploit the simple topology of a trace to do very powerful data-flow based redundancy elimination

Number/Percentage of Ops Removed by PyPy Optimization

	num loops	new	removed	get/set	removed	guard	removed	all ops	removed
crypto_pyaes	78	3088	50%	57148	25%	9055	95%	137189	80%
django	51	673	54%	19318	18%	3876	93%	55682	85%
fannkuch	43	171	49%	886	63%	1159	81%	4935	45%
go	517	12234	76%	200842	21%	53138	90%	568542	84%
html5lib	498	14432	68%	503390	11%	71592	94%	1405780	91%
meteor-contest	59	277	36%	4402	31%	1078	83%	12862	68%
nbody	13	96	38%	443	69%	449	78%	2107	38%
pyflate-fast	162	2278	55%	39126	20%	8194	92%	112857	80%
raytrace-simple	120	3118	59%	91982	15%	13572	95%	247436	89%
richards	87	844	4%	49875	22%	4130	91%	133898	83%
spambayes	314	5608	79%	117002	11%	25313	94%	324125	90%
spectral-norm	38	360	64%	5553	20%	1122	92%	11878	77%
telco	46	1257	90%	37470	3%	6644	99%	98590	97%
twisted-names	214	5273	84%	100010	10%	23247	96%	279667	92%
total	2240	49709	70%	1227447	14%	222569	93%	3395548	89%

Such degree of allocation removal was not seen in any general-purpose JIT

PEPM 2011: Allocation Removal by Partial Evaluation in a Tracing JIT

Common Pitfalls of Existing Reusing JIT Approaches

1. Over-reliance on the JIT alone to improve the performance and underestimating the importance of optimizing the runtime

   For example, a) optimizing named lookup by analyzing hashtable implementations vs. b) implementing named lookup as hidden classes and using runtime feedback to them to indexed lookup

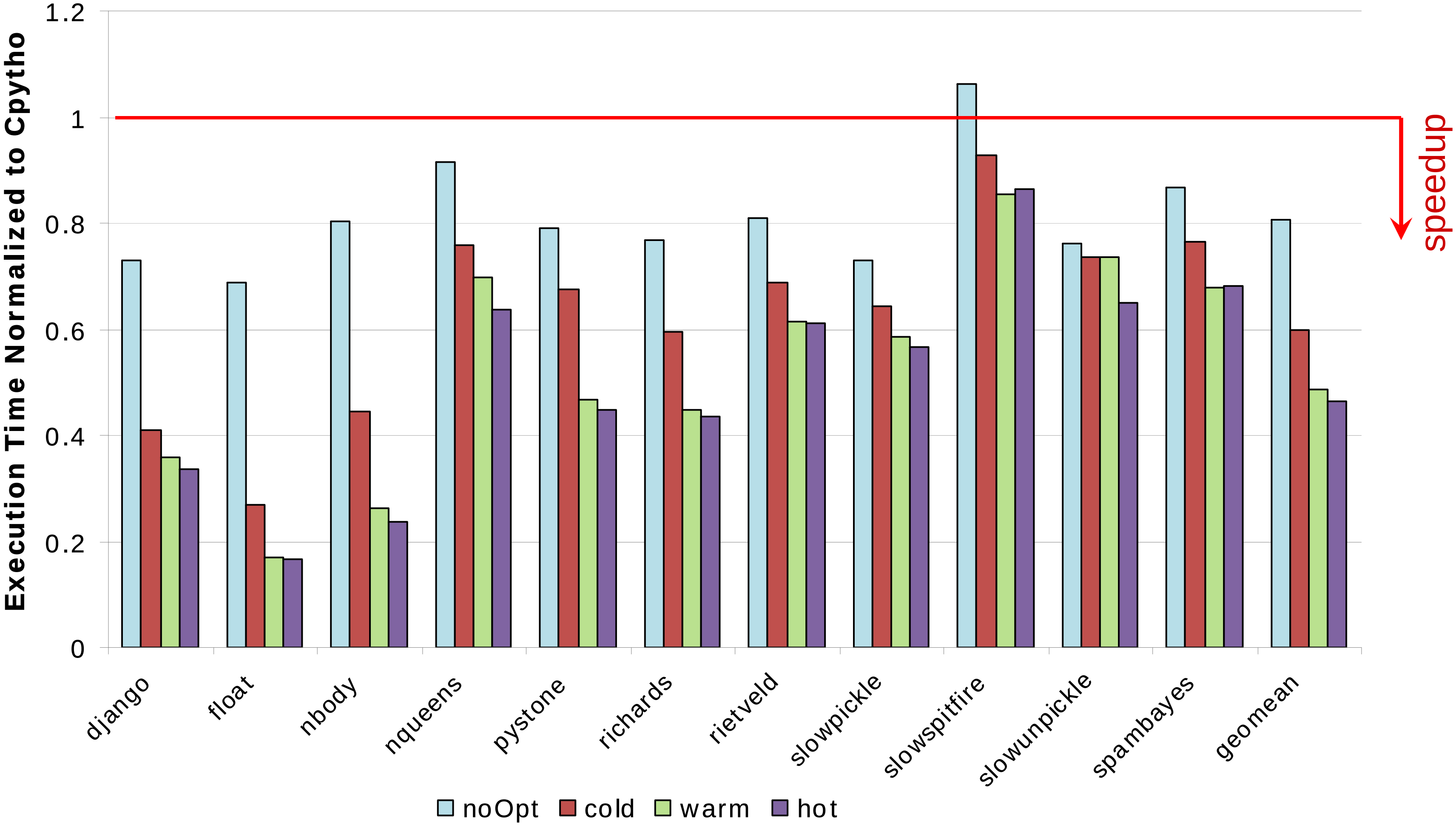
2. Over-reliance on traditional redundancy elimination optimizations to reduce path length of the fat runtime

   Fat runtime imposes two major hurdles to effective dataflow

   - ❑ Long call-chain requires excessive inlining capacity
   - ❑ Excessive redundant heap operations

3. Not emphasizing enough on, **specialization**, a unique and abundant optimization opportunity in scripting language runtime

Effect of Different Optimization Levels: Fiorano JIT



Tips for Reusing JIT Designers

1. Understand characteristics of your runtime
   - identify dominant operations w/ high overhead
   - understand the nature of excessive computation (e,g, heap, branch, call)

2. Remove excessive path lengths in the runtime as much as possible

3. Inside the reusing JIT, focus on the JIT's ability to specialize

4. Boosting existing optimizations in reusing JIT

Typical Profile of a "Fat" Scripting Language Runtime

Instruction path length profile of a typical Python bytecode in Jython runtime

# Java Bytecode	Instruction path length per python bytecode				
	LOAD_LOCAL	BINARY_ADD (int+int)	LOAD_ATTR (self.x)	COMPARE (int > 0)	CALL_FUNCT (self.op())
heap-read	3	5	29	17	53
heap-write	0	2	4	2	16
heap-alloc	0	1	1	0	2
branch	2	8	19	18	34
invoke (JNI)	0	17(0)	23(0)	26(2)	23(2)
return	0	17	23	26	23
arithmetic	0	5	38	8	11
local/const	6	60	152	96	154
Total	**12**	**115**	**289**	**191**	**313**

CPython runtime exhibits similar characteristics

Tips for Reusing JIT Designers

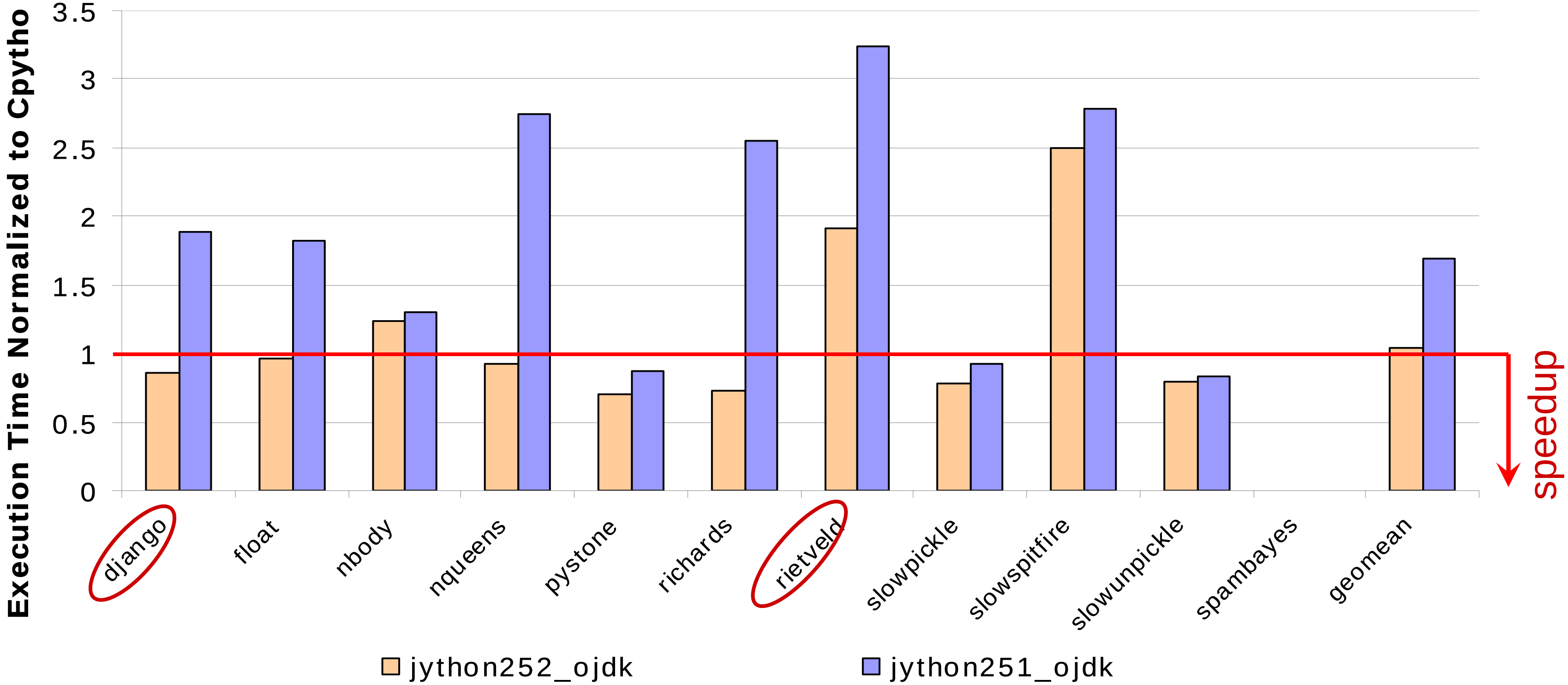
1. Understand characteristics of your runtime
   - identify dominant operations w/ high overhead
   - understand the nature of excessive computation (e,g, heap, branch, call)

2. Remove excessive path lengths in the runtime as much as possible
   - adopt best practice of VM implementation
   - re-evaluate the improved runtime (Step 1)

3. Inside the reusing JIT, focus on the JIT's ability to specialize

4. Boosting existing optimizations in reusing JIT

Effect of Runtime Improvement: Jython 2.5.1 to 2.5.2



- Improvements from Jython 2.5.1 to 2.5.2
  - more than 50% reduction in path length of CALL_FUNCTION
  - significant speedups on large benchmarks with frequent calls

Tips for Reusing JIT Designers

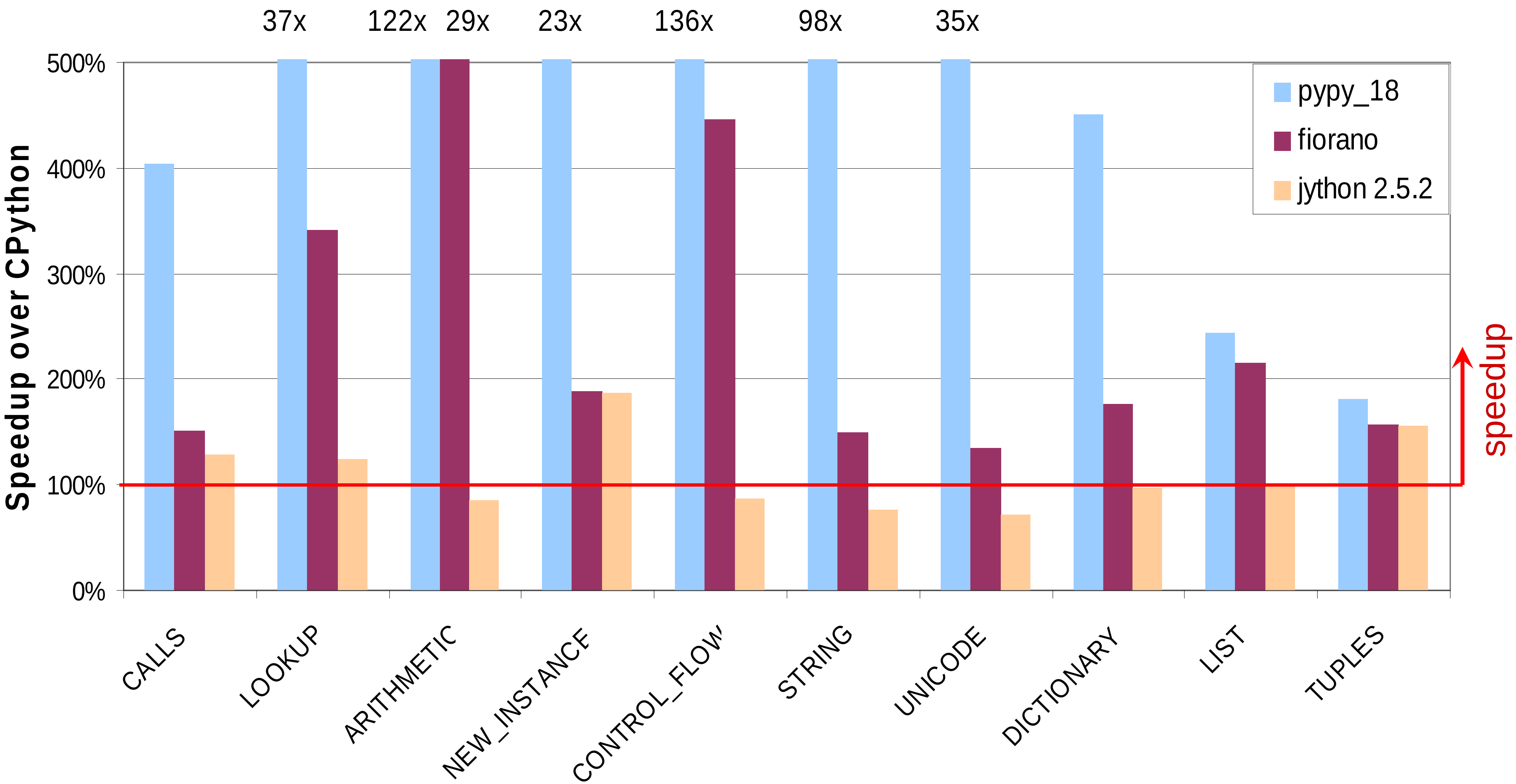
1. Understand characteristics of your runtime
   - identify dominant operations w/ high overhead
   - understand the nature of excessive computation (e,g, heap, branch, call)

2. Remove excessive path lengths in the runtime as much as possible
   - adopt best practice of VM implementation
   - re-evaluate the improved runtime (Step 1)

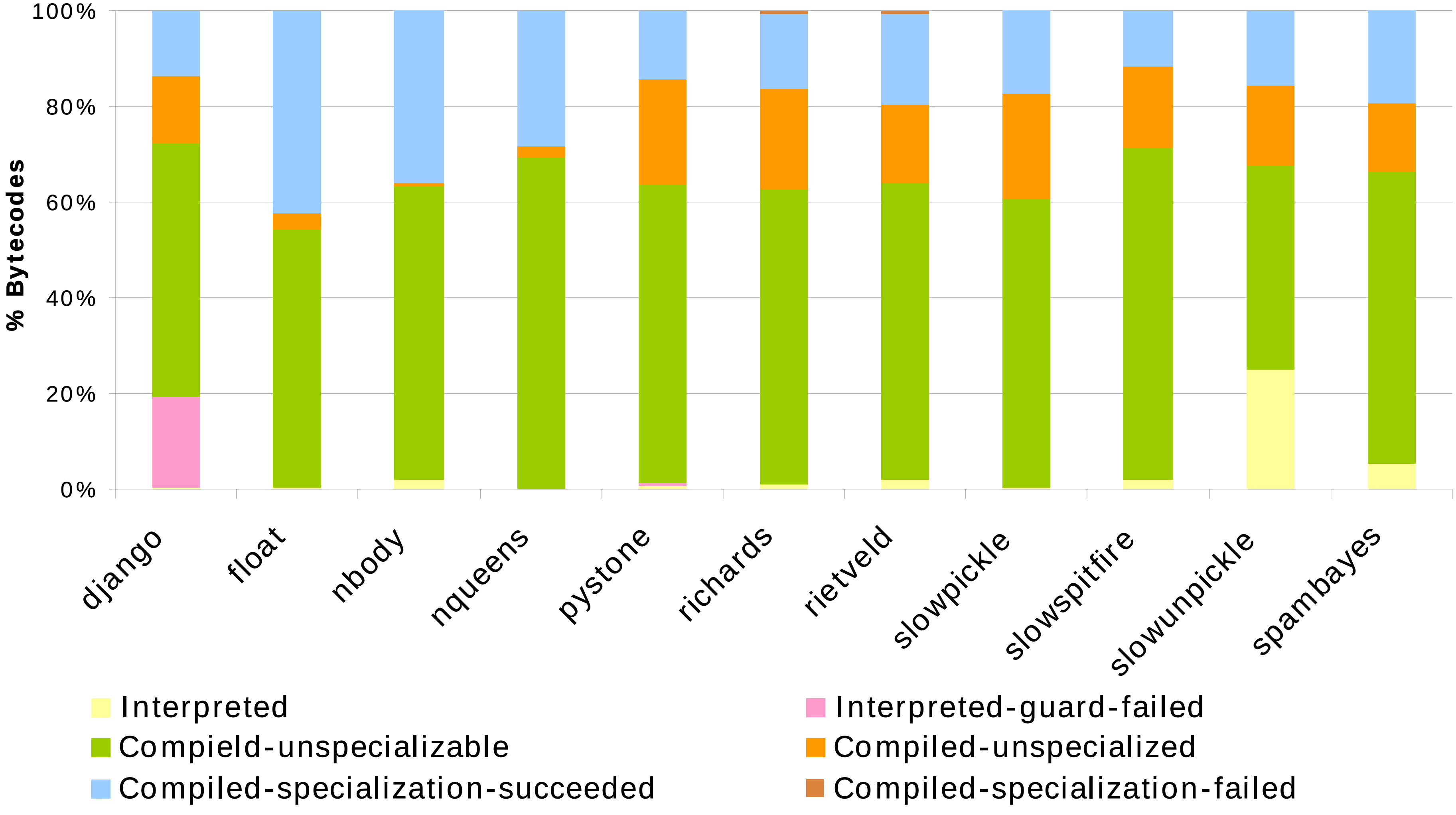
3. Inside the reusing JIT, focus on the JIT's ability to specialize
   - Coverage: how many are specialized and specialized successfully
   - Degree of strength reduction: how fast is the fast version of specialization

4. Boosting existing optimizations in reusing JIT

Pybench: Speedup of JITs on Common Python Idioms



Breakdown of Dynamic Python Bytecode Execution



Tips for Reusing JIT Designers

1. Understand characteristics of your runtime
   - identify dominant operations w/ high overhead
   - understand the nature of excessive computation (e,g, heap, branch, call)

2. Remove excessive path lengths in the runtime as much as possible
   - adopt best practice of VM implementation
   - re-evaluate the improved runtime (Step 1)

3. Inside the reusing JIT, focus on the JIT's ability to specialize
   - Coverage: how many are specialized and specialized successfully
   - Degree of strength reduction: how fast is the fast version of specialization

4. Boosting existing optimizations in reusing JIT

Effective Boosting Techniques in Fiorano JIT

- ❑Runtime feedback driven specialization
  - Types are typically quite stable to rely on simple runtime feedback
  - Achieve much higher coverage than analysis based approach

- ❑Focus on ***early*** path length reduction, especially during translation to IR

- ❑Guard-based specialization
  - Compared to versioning based specialization, guard eliminates data-flow join
  - Need to monitor guard failure and need de-optimization support

Concluding Remarks

- Whenever an interpreted language emerges, reusing an existing JIT (LLVM, Java JIT) to compile the language becomes an economic option

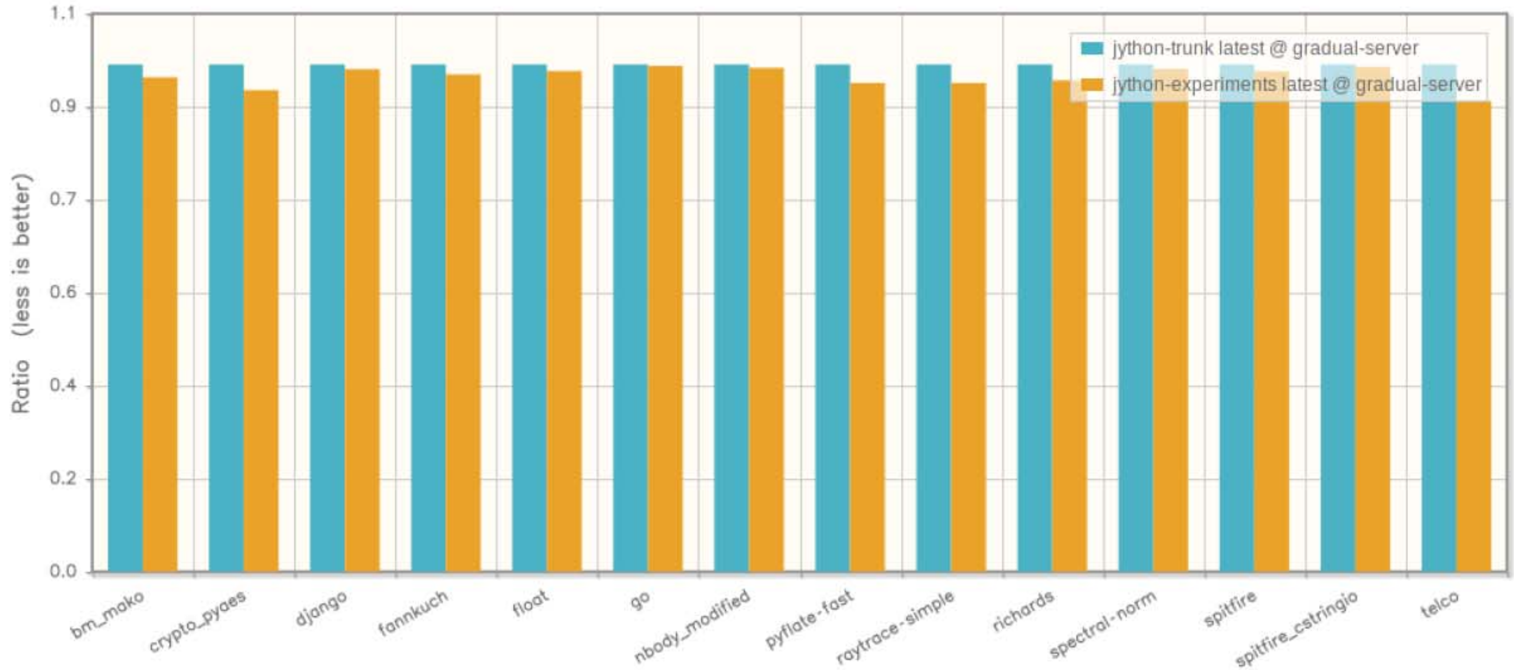
- Many reusing JITs for scripting languages do not live up to the expectation. Why?
  - The root cause of scripting language overhead is the excessive path length explosion in the language runtime (10~100x compared to static language)
  - Traditional JITs are **not** capable of massive path length reduction in language runtime permeated with heap/pointer manipulation and control-flow join

- We offer lessons learned and recommendations to reusing JITs designers
  - Focus on path length reduction as the primary metrics to design your system
  - Do not solely rely on the JIT, improving the language runtime is as important
  - When reusing optimizations in the JIT, less is more
  - Instead, focus on specialization, runtime feedback, and guard-based approach

❑BACK UP

InvokeDynamics and JVM Languages

Results: 4% improvement across the suite



Performance of pilot implementation of Jython using invokedynamics

By Shashank Bharadwaj, University of Colorado
http://wiki.jvmlangsummit.com/images/8/8d/Indy_and_Jython-Shashank_Bharadwaj.pdf

Evolution of Javascript JITs

- Google
  - V8:
    - efficient object representation
    - hidden classes
    - GC
  - Crankshaft: “traditional” optimizer (Dec 2010)
    - adaptive compilation
    - aggressive profiling
    - optimistic assumptions
    - SSA, invariant code motion, register allocation, inlining
    - Overall, improved over V8 by 50%
  - Beta release of Chrome with native client integrated
    - C/C++ codes executed inside browser with security restrictions close to Javascripts
- Mozilla
  - TraceMonkey
    - trace-JIT, aggressive type specialization
  - JaegerMonkey (Sept, 2010, Firefox 4)
    - method-JIT, inlining
  - IonMonkey (2011)
- Apple
  - Nitro JIT (Safari 5)
  - “ 30% faster than Safari 4, 3% faster than Chrome 5, 2X faster than Firefox 3.6”
- Microsoft
  - Chakra JIT (IE9)
    - async compilation
    - type optimization
    - fast interpreter
    - library optimization

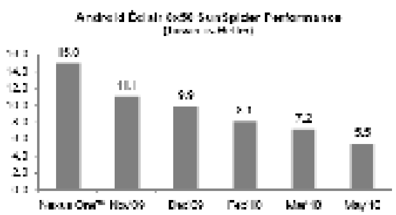
JIT compilation for Javascript is a reality

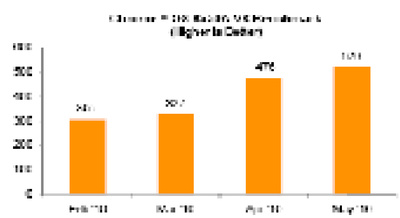
- all major browser/mobile vendors have their own Javascript engine!
- Nodejs: server-side Javascript using asynchronous event driven model

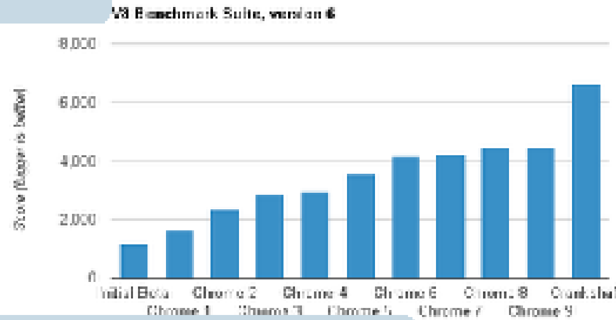
JavaScript JIT Compilation: Big Opportunity for Highly Impacting Contribution to Industry

Very fast evolution lately!

V8 Performance Optimizations





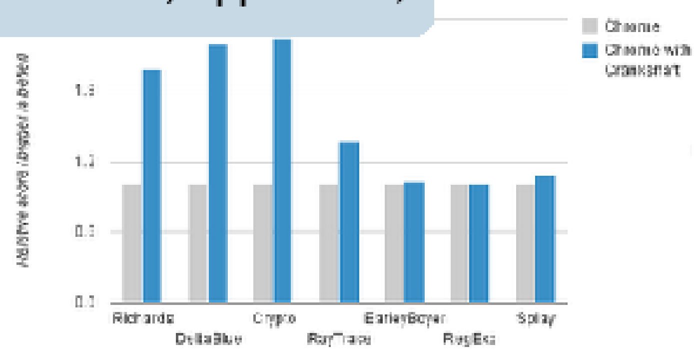


Safari Performance

iOS 4.3

Surf at the speed of fast.

New sophisticated optimizers (e.g. V8 Crankshaft, Apple Nitro)



- Accelerated evolution in the last few years
- Highly competitive

= Big Opportunity for Compiler Research! (also to define a more efficient language…)

Compilation for Mobile

ST ERICSSON

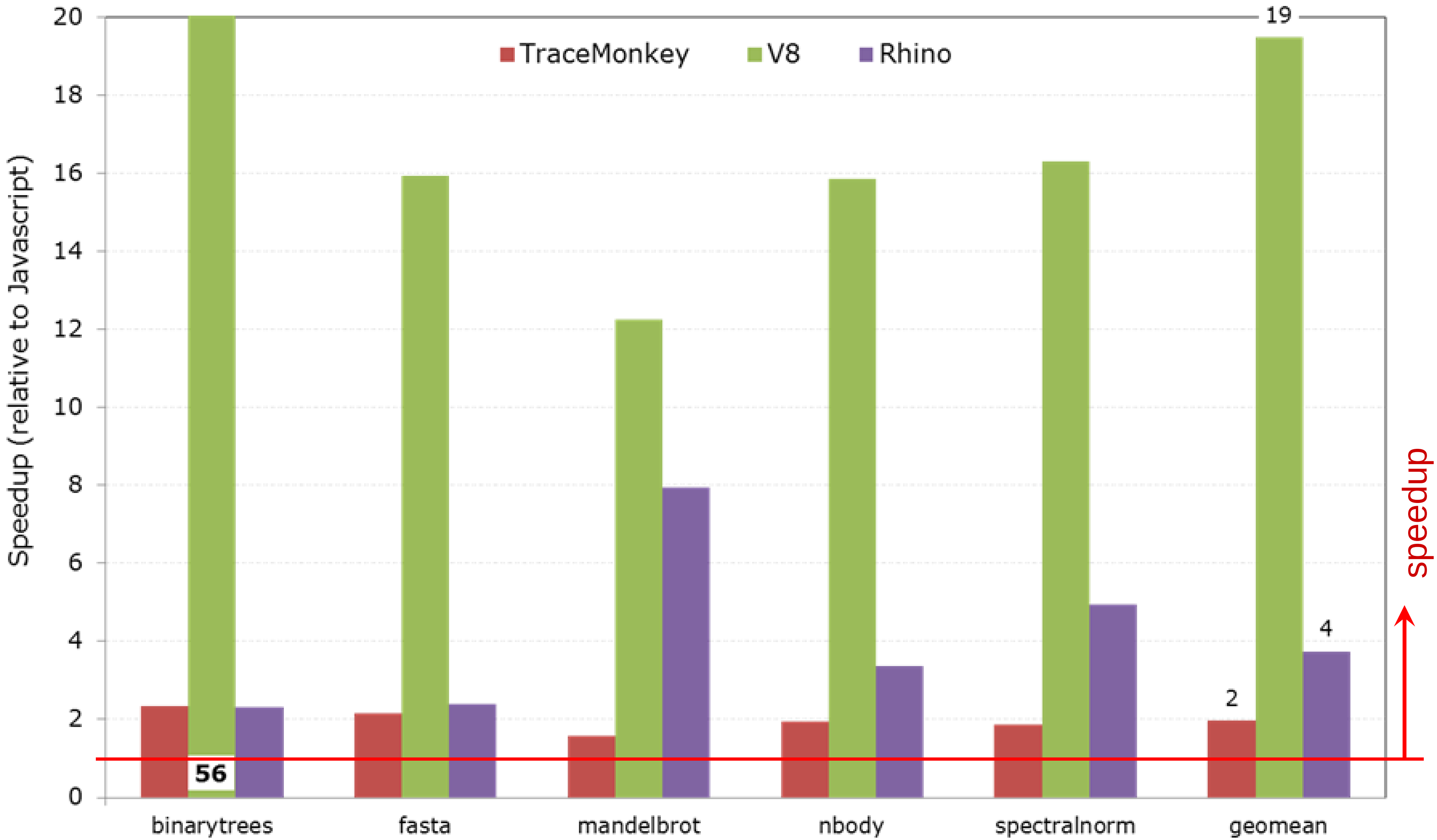
Marco Cornero (ST Ericsson): http://www.hipeac.net/system/files/2011-04-06_compilation_for_mobile.pdf

Google: Crankshaft JIT

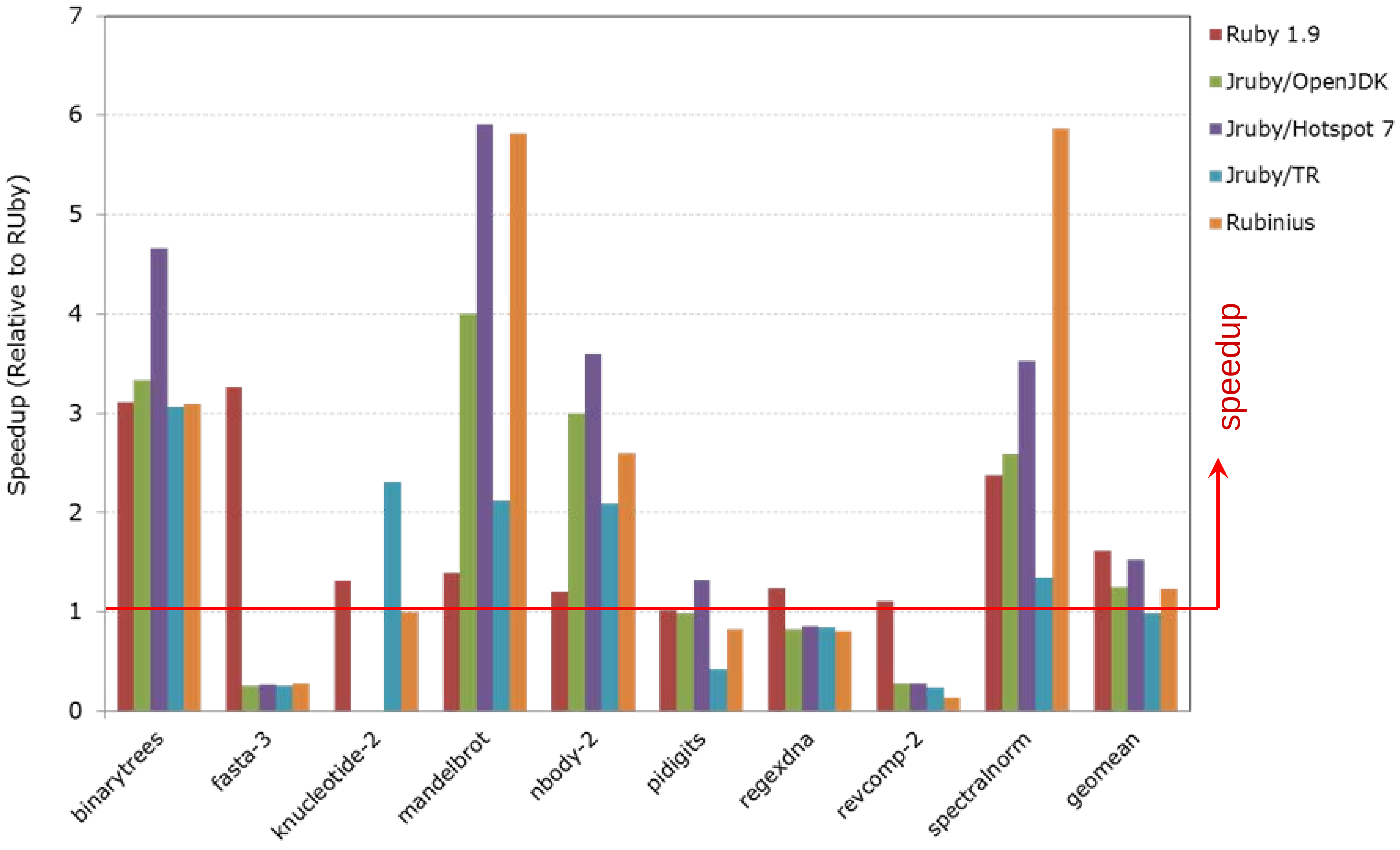
- ❑A new JIT compiler for V8 (Dec 2010)
  - – Performance improvement by 50%, upto 2X (V8 benchmark)
  - – Mostly benefits codes with hot loops, not for very short scripts (SunSpider)
  - – Improved start-up time for web apps, e.g., gmail

- ❑Crankshaft JIT (adaptive compilation):
  - – Base compiler: simple code generation
  - – Runtime profiler: identify hot codes and collect type info
  - – Optimizing compiler (hot codes only): SSA, loop invariant code motion, linear-scan RA, inlining, using runtime type info
  - – Deoptimization support: can bail out of optimized codes if runtime assumption (e.g., type) is no longer valid

Performance of Javascript implementations



Performance of Ruby Implementations



IronPython: DynamicSites

- Optimize method dispatch (including operators)
- Incrementally create a cache of method stubs and guards in response to VM queries

```
public static object Handle(object[],
  FastDynamicSite<object, object, object> site1,
  object obj1, object obj2) {
    if (((obj1 != null) && (obj1.GetType() == typeof(int)))
       && ((obj2 != null) && (obj2.GetType() == typeof(int)))) {
       return Int32Ops.Add(Converter.ConvertToInt32(obj1),
                           Converter.ConvertToInt32(obj3));
    }
    if (((obj1 != null) && (obj1.GetType() == typeof(string)))
       && ((obj2 != null) && (obj2.GetType() == typeof(string)))) {
       return = StringOps.Add(Converter.ConvertToString(obj1),
                              Converter.ConvertToString(obj2));
    }
    return site1.UpdateBindingAndInvoke(obj1, obj3);
}
```

- Propagate types when UpdateBindingAndInvoke recompiles stub