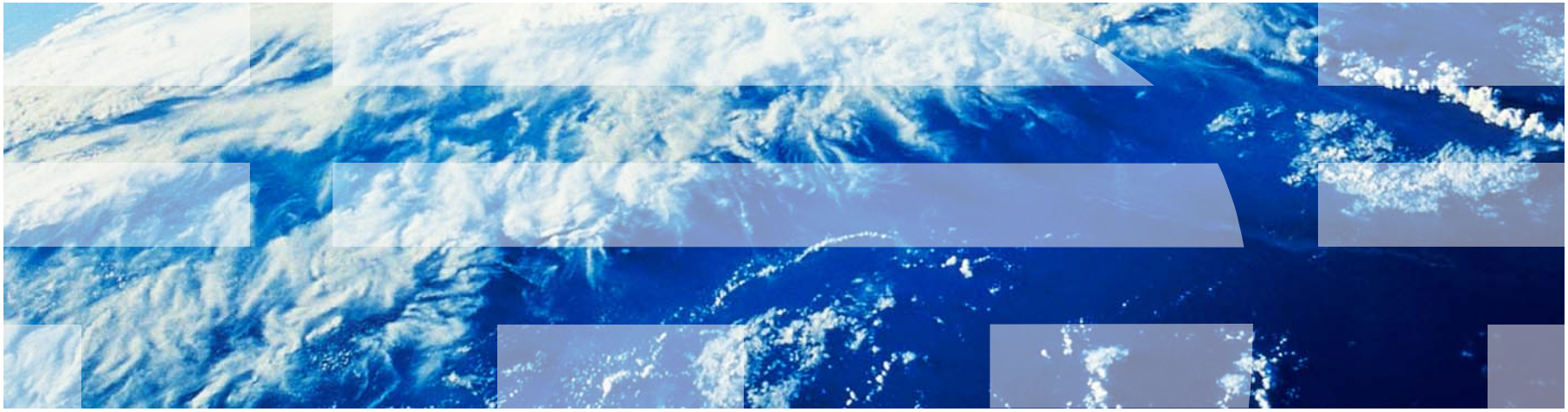
IBM

Compilers are from Mars, Dynamic Scripting Languages are from Venus

Peng Wu
IBM Research



© 2010 IBM Corporation

Compilation for Dynamic Scripting Languages

- Trend in emerging programming paradigms
  - **Dynamic scripting languages** (DSLs) are gaining popularity, and start to be used for production development

Commercial deployment
- Facebook (PHP)
- YouTube (Python)
- Invite Media (Python)
- Twitter (Ruby on Rails + Scala)
- ManyEyes (Ruby on Rails)

Cloud
- Google AppEngine (Python)

TIOBE Language Index

Rank	Name	Share
1	C	16.986%
2	Java	16.668%
3	PHP	10.298%
4	C++	8.554%
5	Basic	6.757%
6	C#	5.444%
7	Python	5.179%
8	Perl	3.474%
9	Ruby	2.370%
10	JavaScript	2.149%

"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

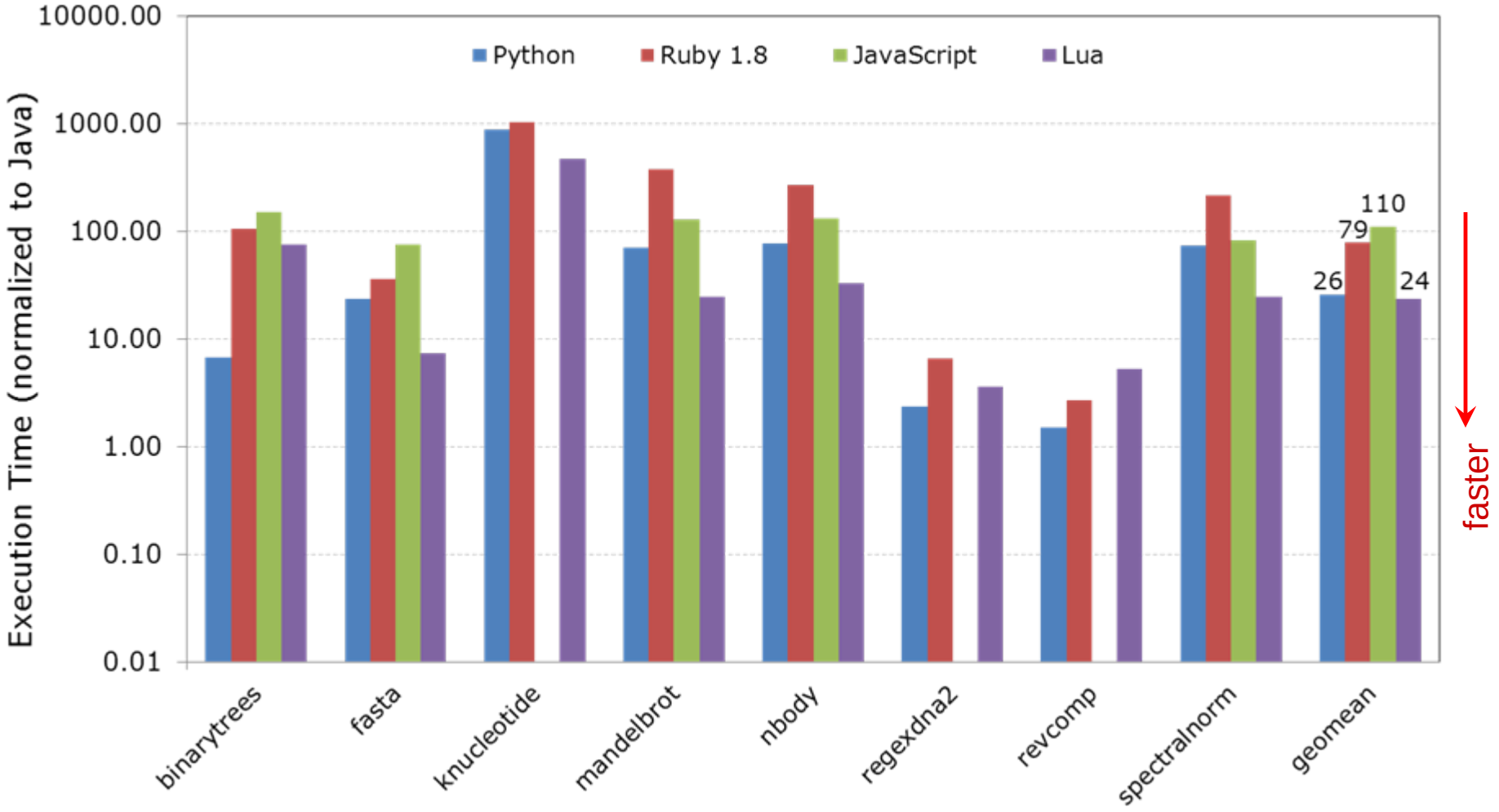
© 2010 IBM Corporation

Motivation

- Dynamic scripting languages (DSL)
  - Python, Ruby, PHP, Javascript, Lua, R, and many others
- Optimization of DSL programs is an active area of research
  - renewed browser wars
    - TraceMonkey (Mozilla), SPUR (MS), V8 (Google)
  - cloud deployment
    - AppEngine from Google
- Significantly slower compared to equivalent in Java and C
  - mostly interpreted, not highly optimized, richer semantics for basic operations
- The research landscape for DSL compilation is vast
  - no low-hanging fruits for compilation
  - a lot of variability in results
  - no agreed principles in the community

© 2010 IBM Corporation

Language 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)

© 2010 IBM Corporation

Python Language and Implementation

- Python is an object-oriented, dynamically typed language
  - also support exception, garbage collection, function continuation

foo.py

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

python bytecode

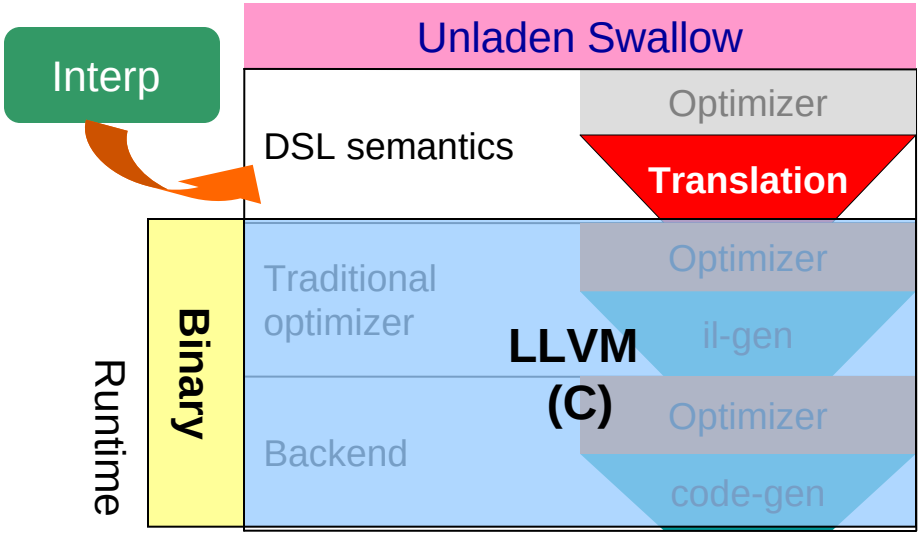
```
 0 LOAD_GLOBAL     0 (len)
 3 LOAD_FAST       0 (list)
 6 CALL_FUNCTION   1
 9 LOAD_CONST      1 (1)
12 BINARY_ADD
13 RETURN_VALUE
```

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

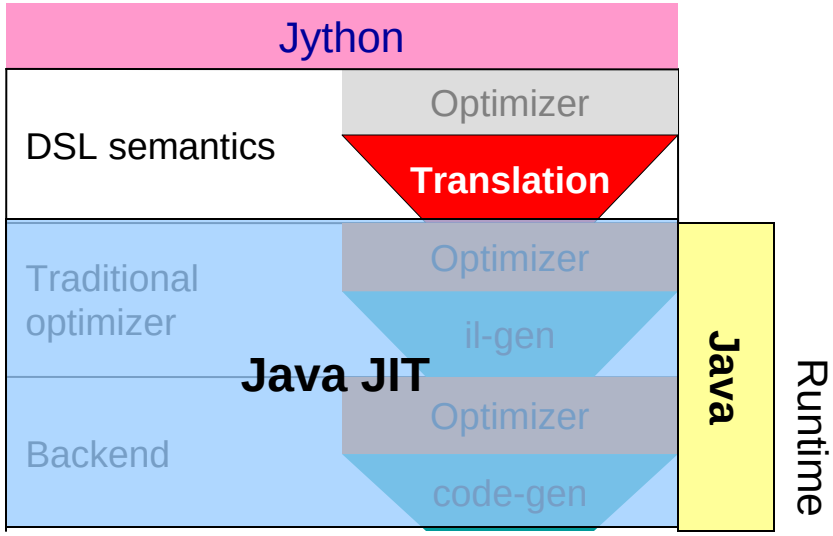
All three involve layers of runtime calls (via function pointers), reference counting, and exception checking

© 2010 IBM Corporation

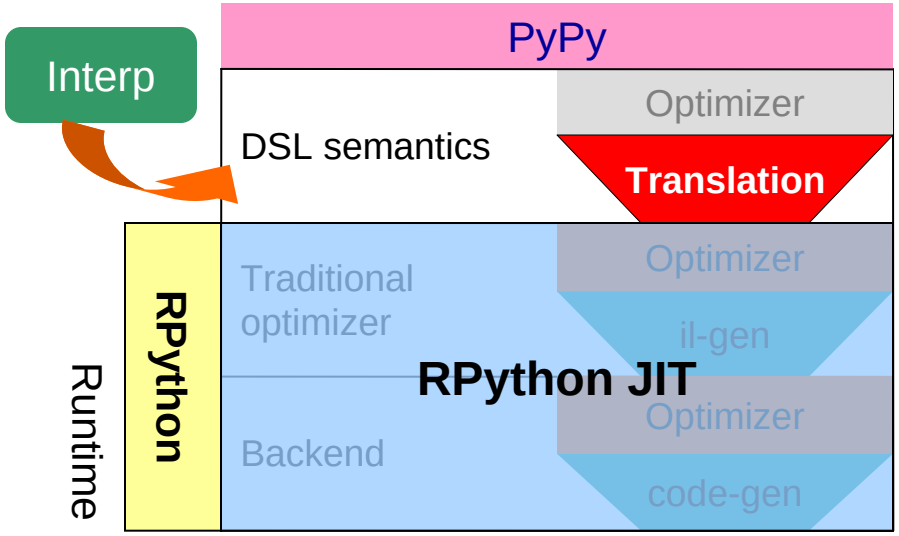
Optimizing Compiler Approaches



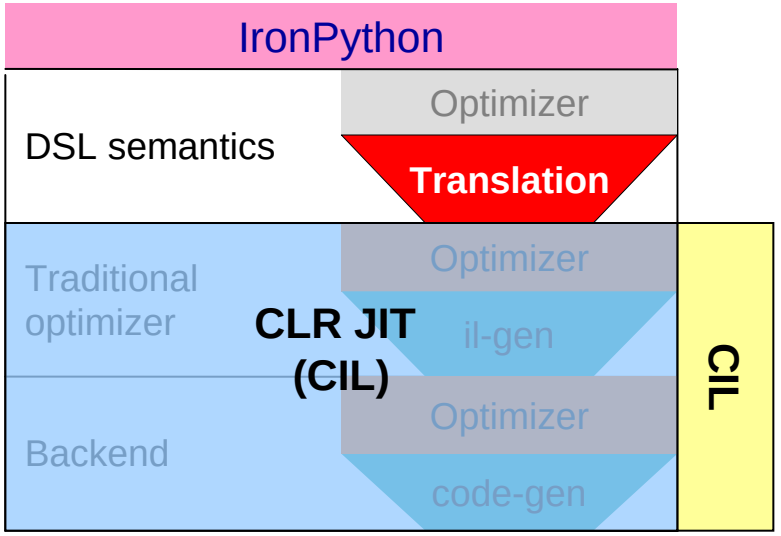
Legacy target language: C/C++



Legacy target language: Java



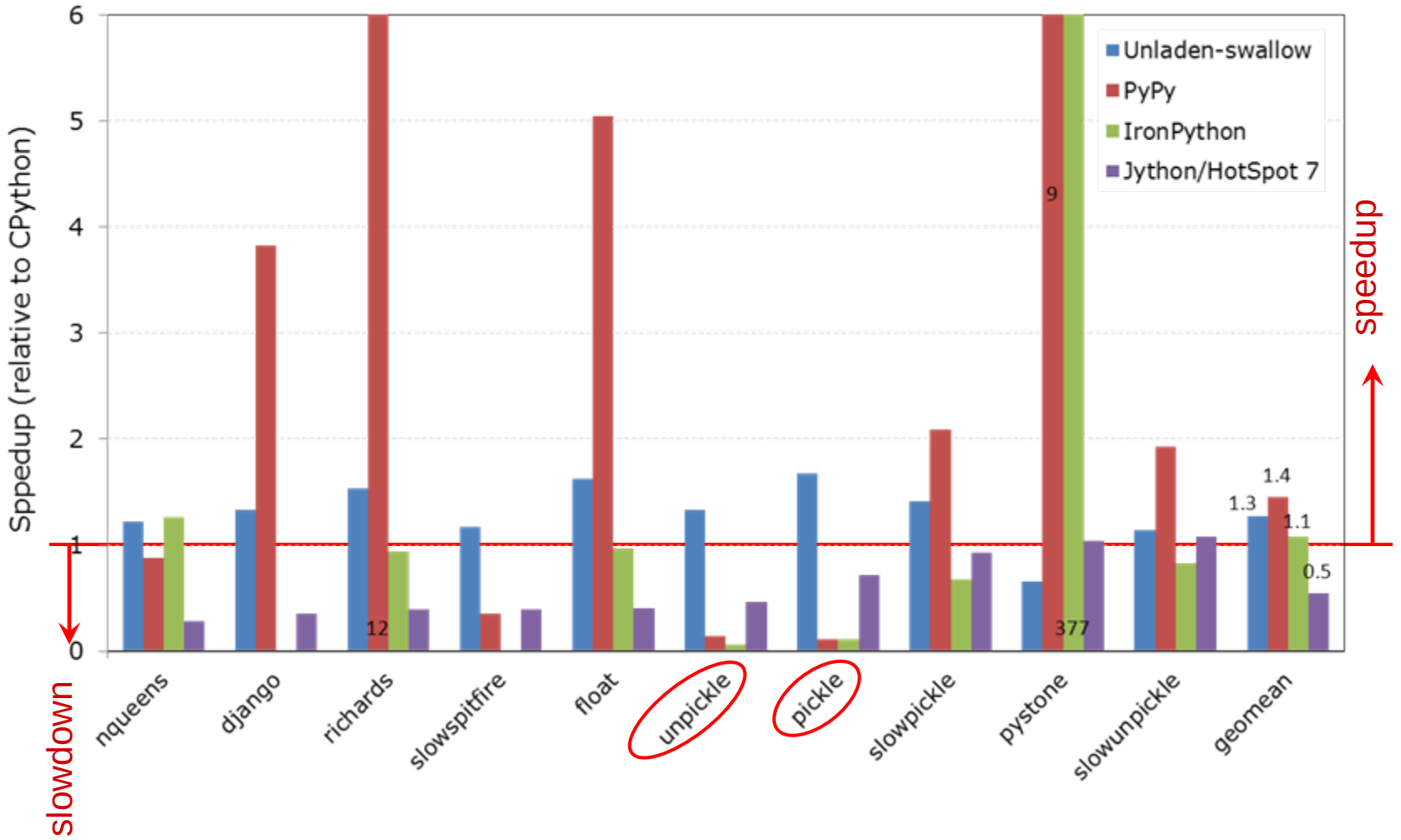
RPython is closest to Python



Legacy target language: C#

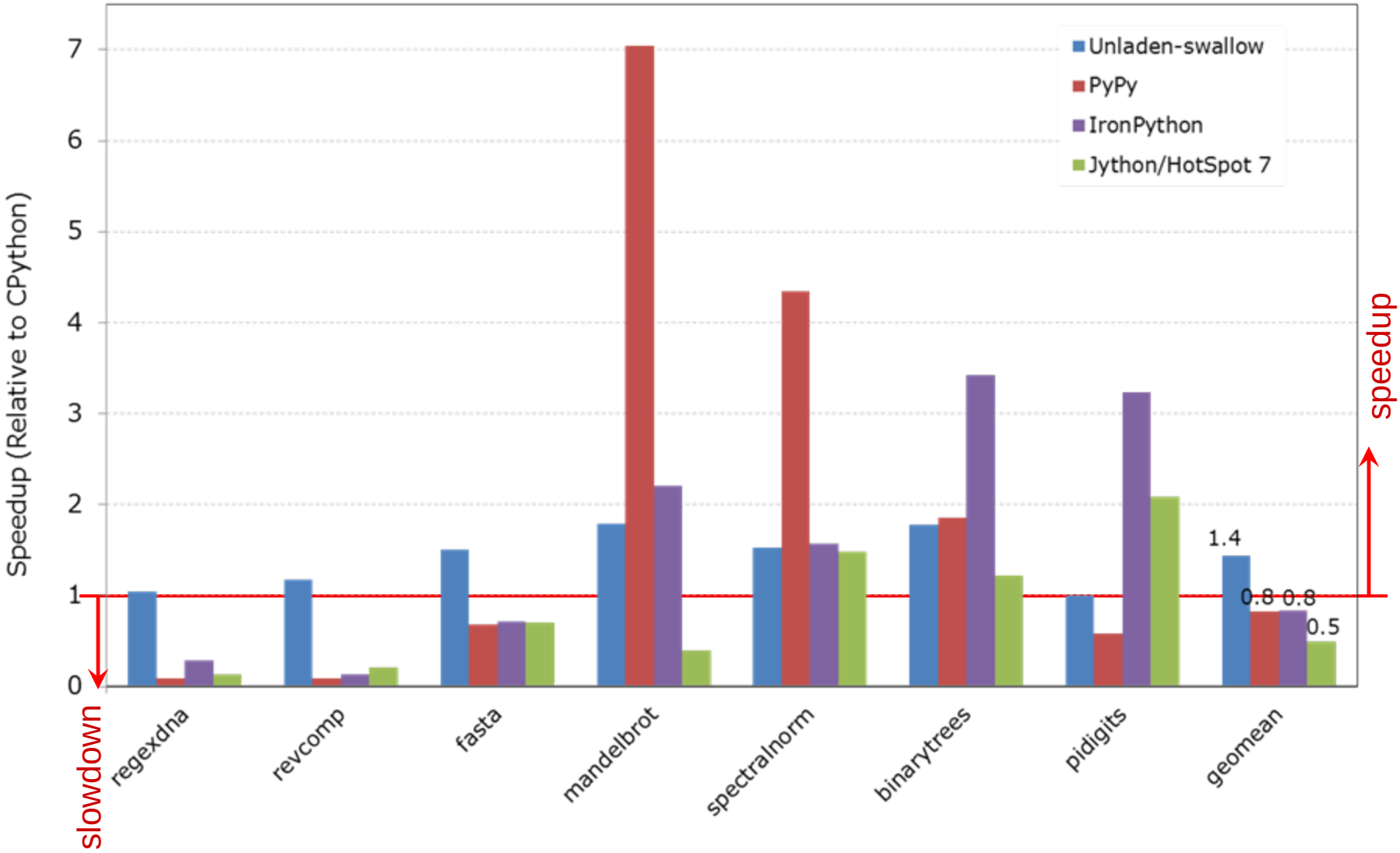
© 2010 IBM Corporation

Python Implementations (Unladen-Swallow benchmarks)



© 2010 IBM Corporation

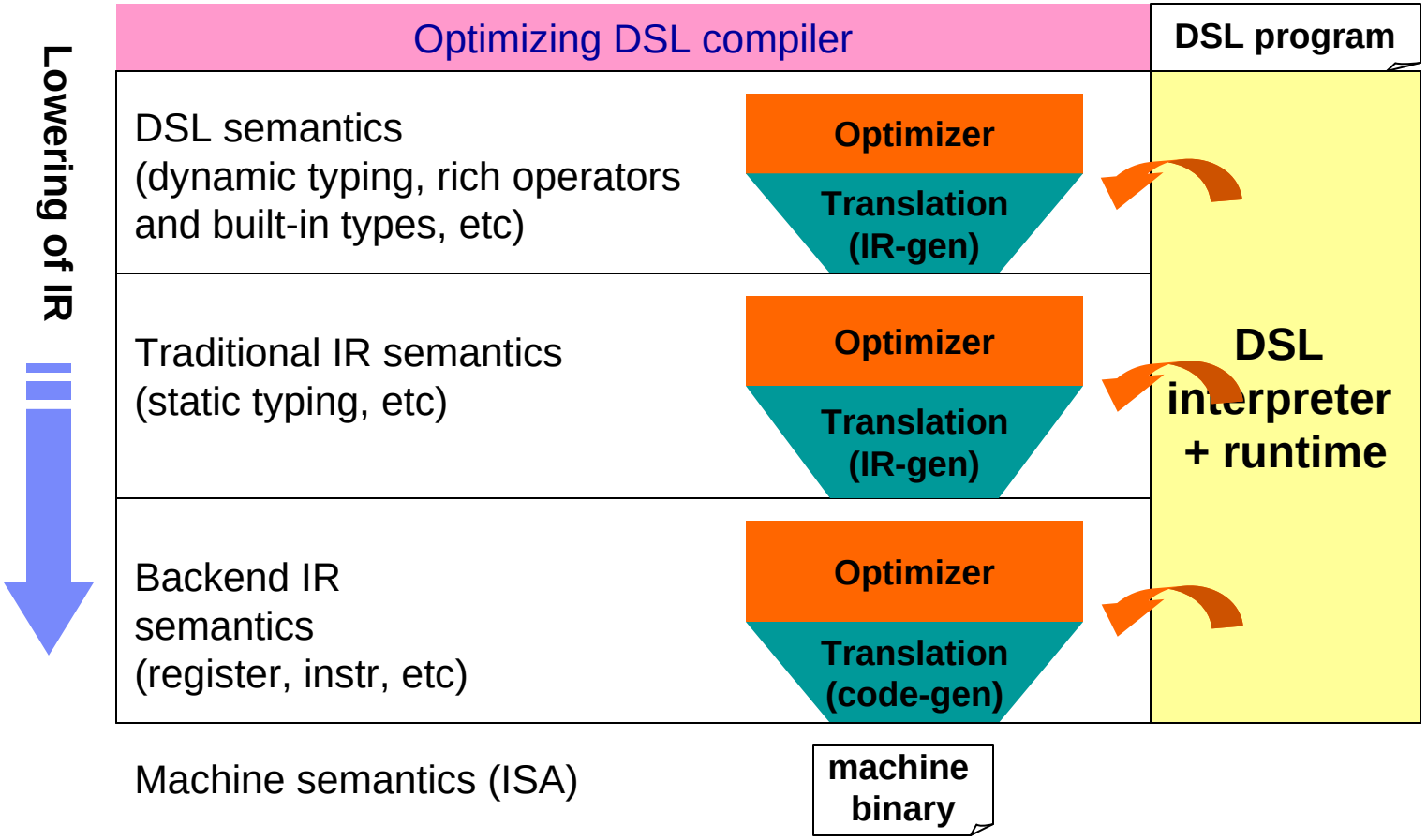
Python Implementations (shootout benchmarks)



Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation

A System View of Optimizing DSL Compilers



© 2010 IBM Corporation

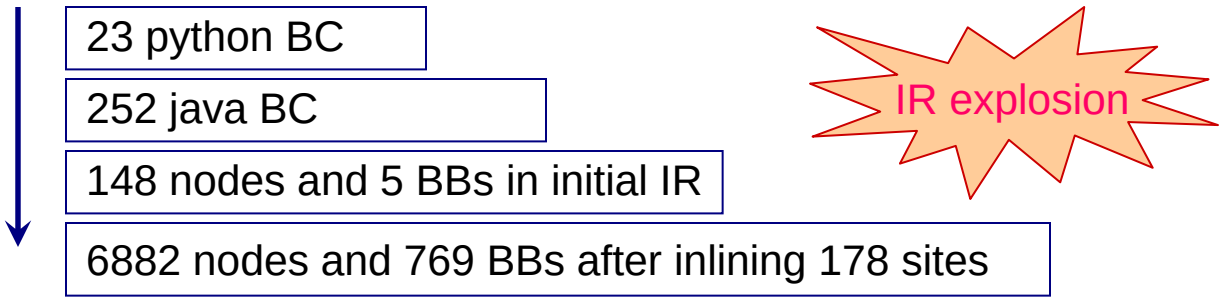
An Optimization Example (LOAD_GLOBAL)

```
100 SETUP_LOOP;
    LOAD_GLOBAL 1 ('foo');
    CALL_FUNCTION;
    …
    JUMP_ABSOLUTE 100;
```

DSL data-flow optimizer (e.g., bytecode optimizer): hoist LOAD_GLOBAL out of the loop *if* one can prove it invariant

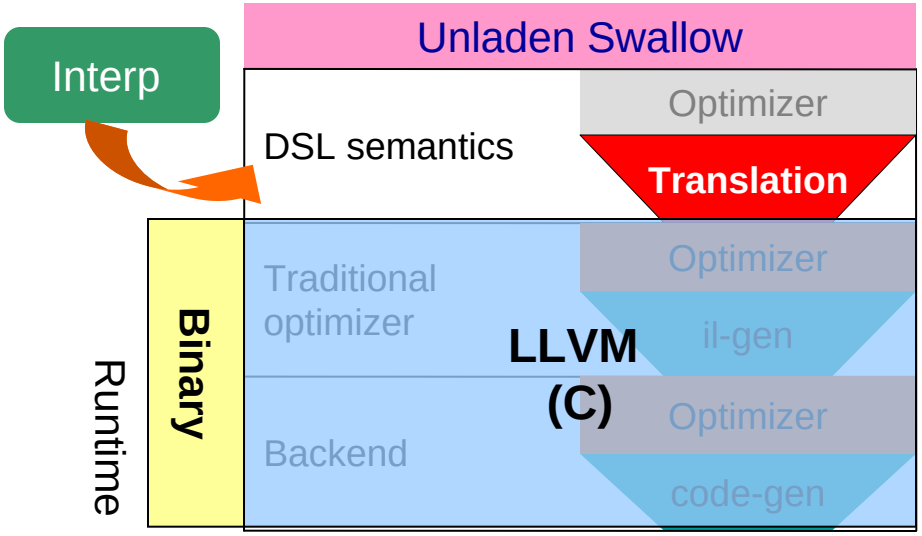
Translation time optimization (e.g., unladen-swallow): in-line caching with guards, 3%~9% improvements on rietfield, django, 2to3 from unladen-Swallow benchmarks.

Optimization inside runtime (e.g., jython): improve dictionary (hashtable) lookup by inlining, code straightening, etc

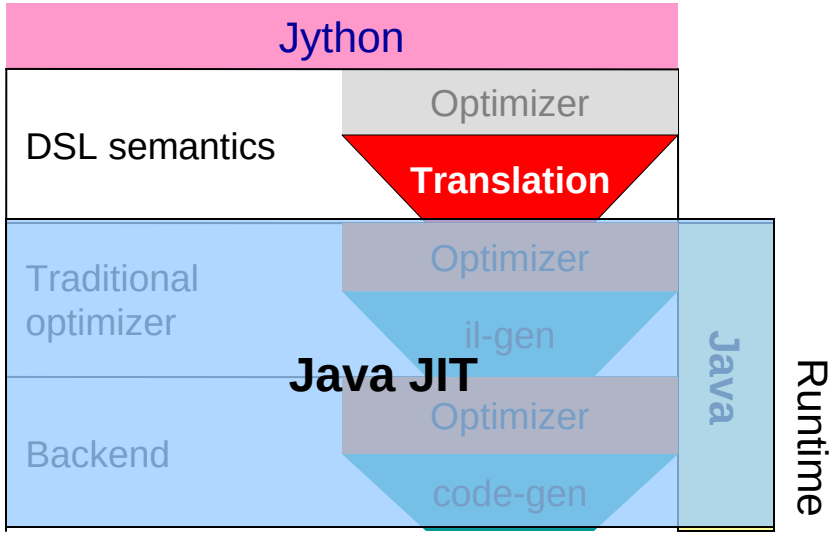


© 2010 IBM Corporation

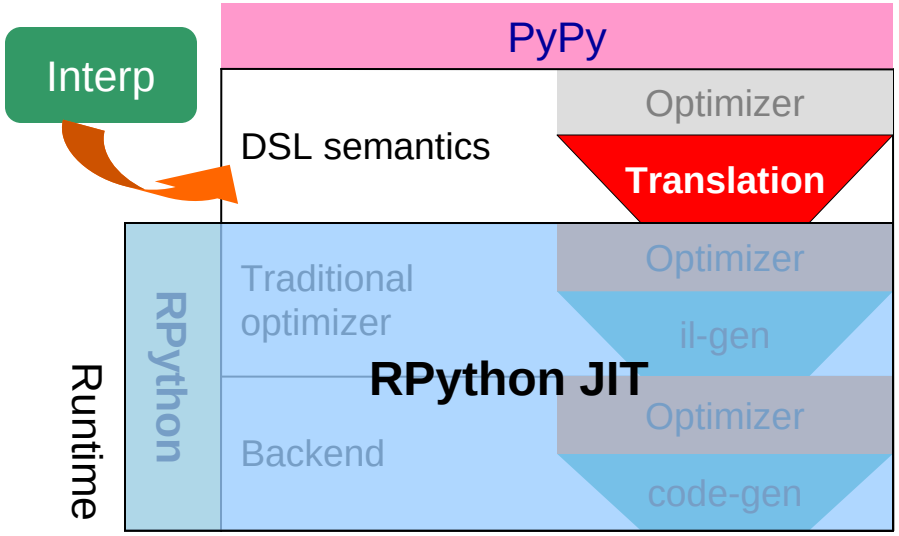
Optimizing Compiler Approaches



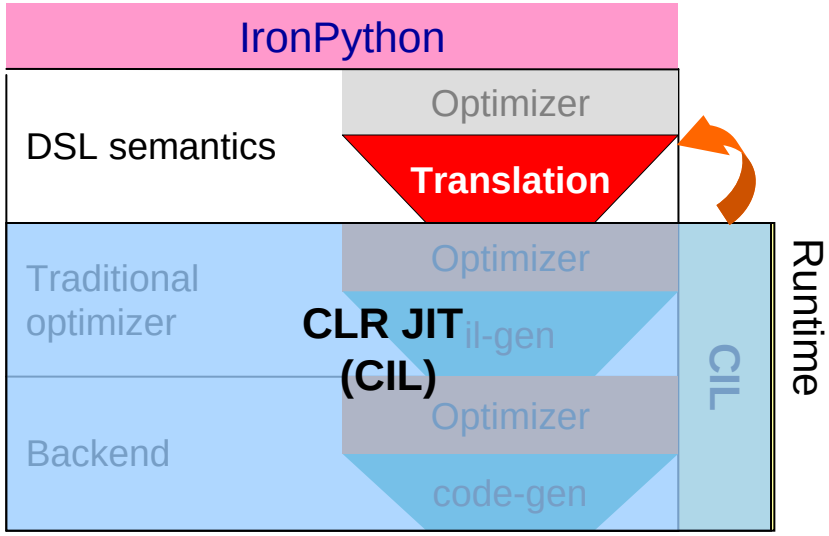
Legacy target language: C/C++



Legacy target language: Java



RPython is closest to Python



Legacy target language: C#

Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation

First Level of Lowering of DSL Semantics

DSL Semantics

Unladen (naïve)	`ref counting; PyBinaryAdd(); error checking;`	LLVM
Jython	`invokevirtual PyObject._add()`	JVM
Unladen w/ feedback	`ref counting; inlined PyIntAdd; error checking;`	LLVM
PyPy	`INT_ADD (unboxed)`	RPython

Direct opt effect

Less dynamism
Less overhead

Semantic loss

LLVM IR
JVM IR
RPython IR

Compact IR

JVM IR
LLVM IR
RPython IR

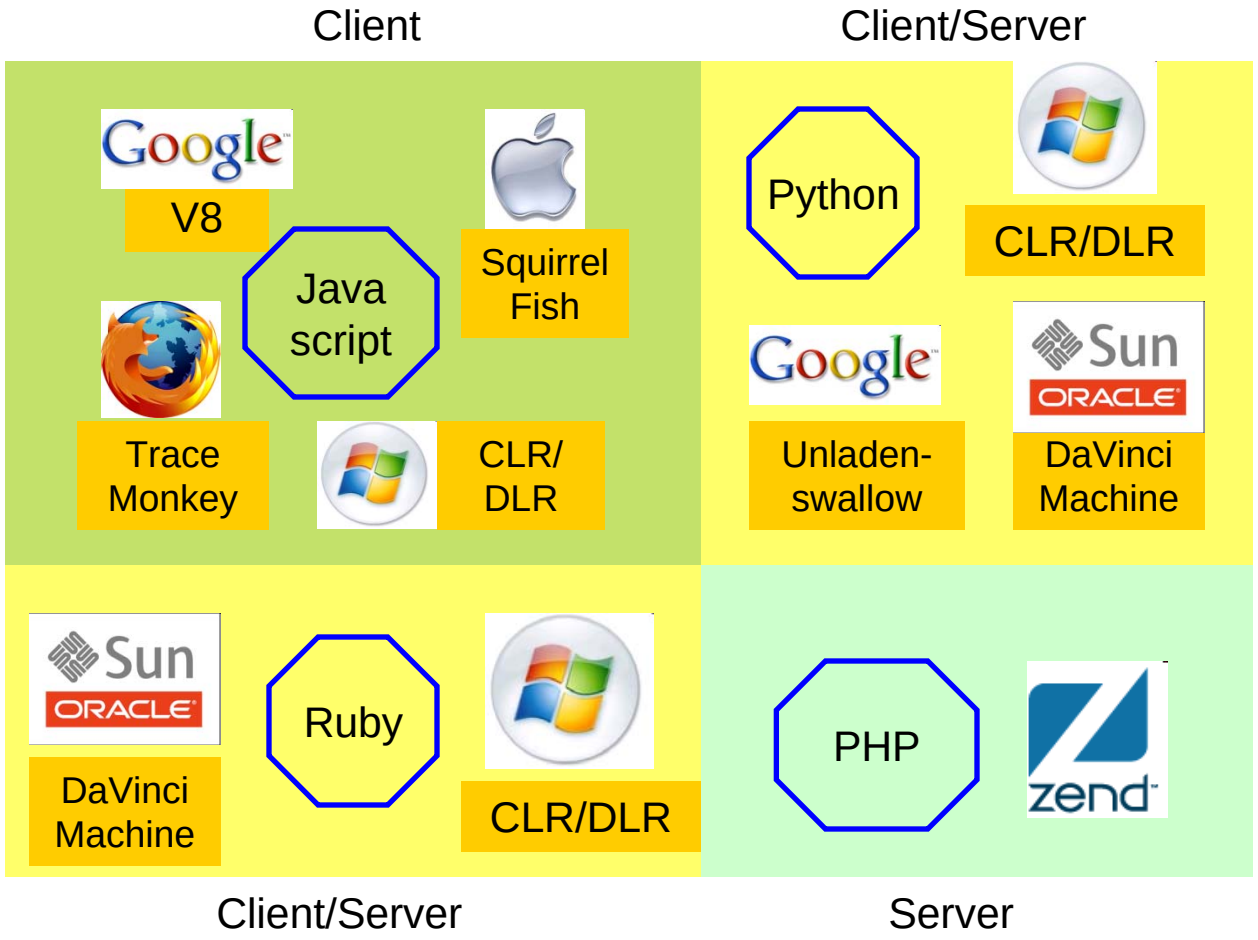
better

impact effectiveness of optimizer at next level

RPython: well-typed, unboxed primitive types, class definition unchange after start-up time

© 2010 IBM Corporation

Dynamic Scripting Language JIT Landscape



- JVM based
  - Jython
  - JRuby
  - Rhino
- CLR based
  - IronPython
  - IronRuby
  - IronJscript
  - SPUR
- Add-on JIT
  - Unladen
  - Rubinius
- Add-on trace JIT
  - PyPy
  - LuaJIT,
  - TraceMonkey
  - SPUR

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

Compilers are from Mars, and Dynamic Scripting Languages are from Venus
© 2010 IBM Corporation

Concluding Remarks (Questions)

The challenges

Dynamically typed languages are slow. And we, the language design & implementation community, does not quite know how to optimize DSLs.

Questions for the compiler community

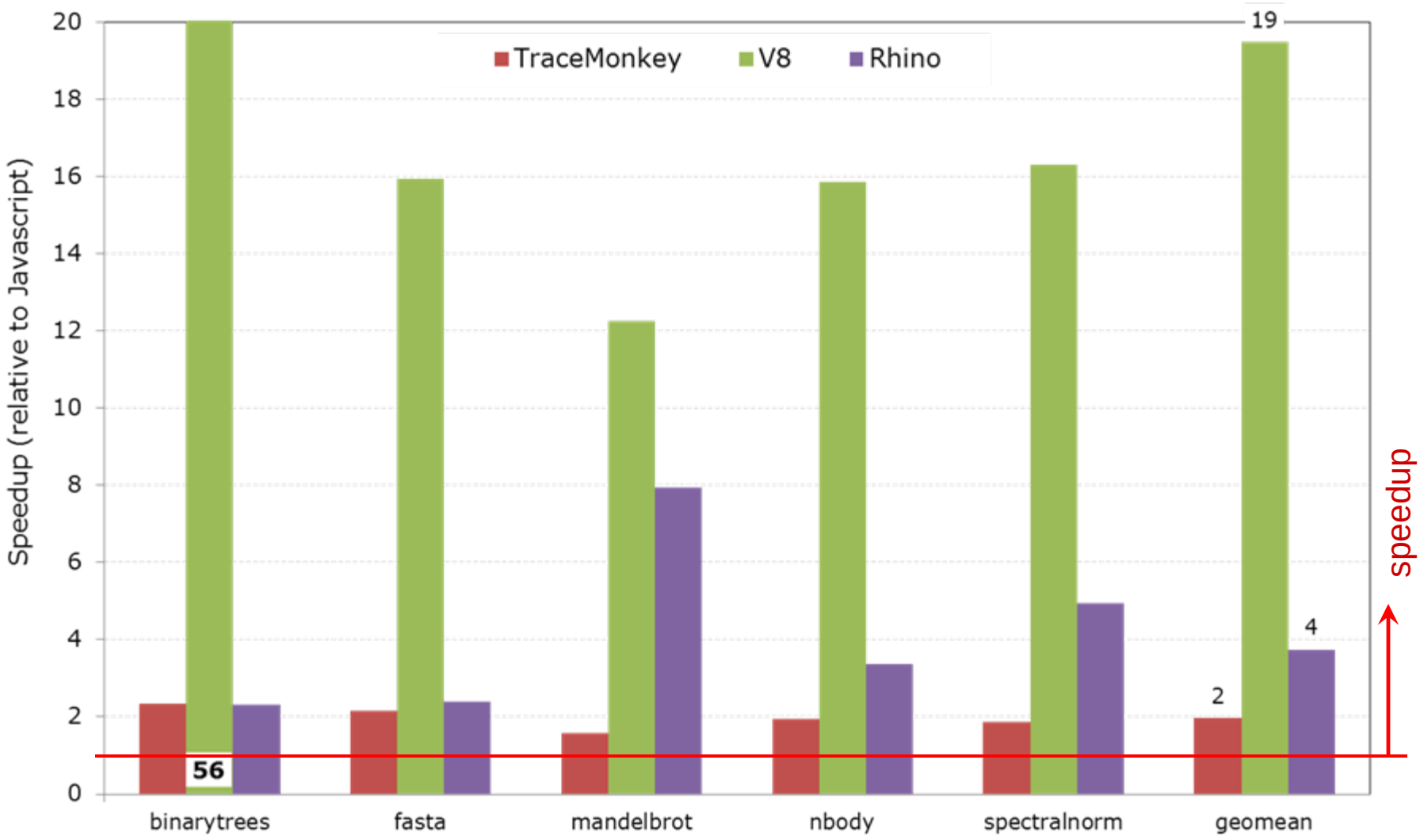
1. What are the right level(s) to optimize dynamic scripting languages?
2. How to introduce DSL semantics into an optimization infrastructure designed for statically typed languages

Our thoughts

- "Naïve" compilation of DSL provides little benefit
- Dynamism and overhead should be reduced at a suitable IR level
  - Semantic lowering can be "*lost-in-translation*" or real "*strength reduction*"
  - Exposing the runtime to optimizer can be double-edged sword: optimizing implementation of the semantics instead of the semantics
- Language runtime needs to be redesigned to maximize optimizer's capability to "strength reduce"

© 2010 IBM Corporation

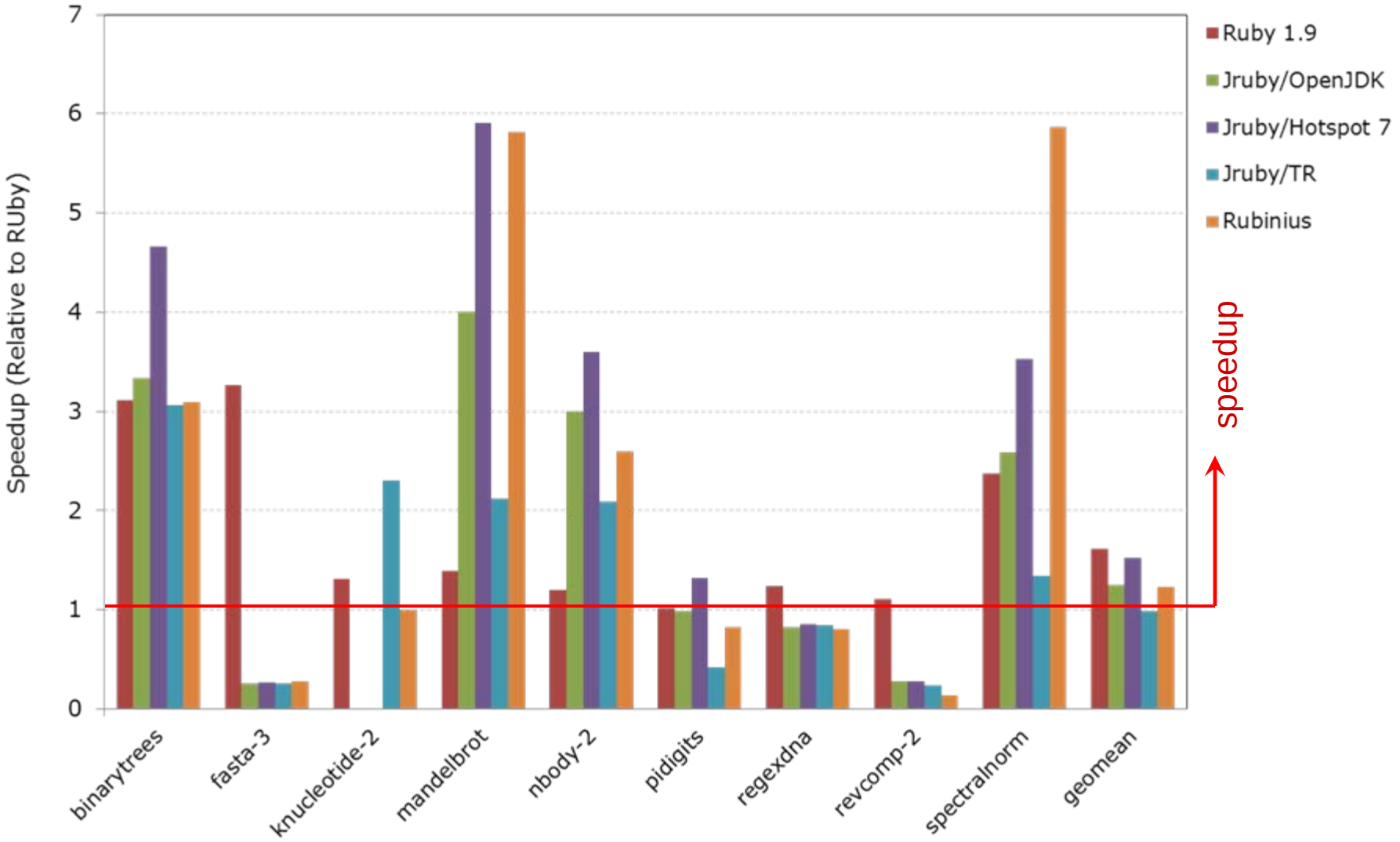
Performance of Javascript implementations



Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation

Performance of Ruby Implementations



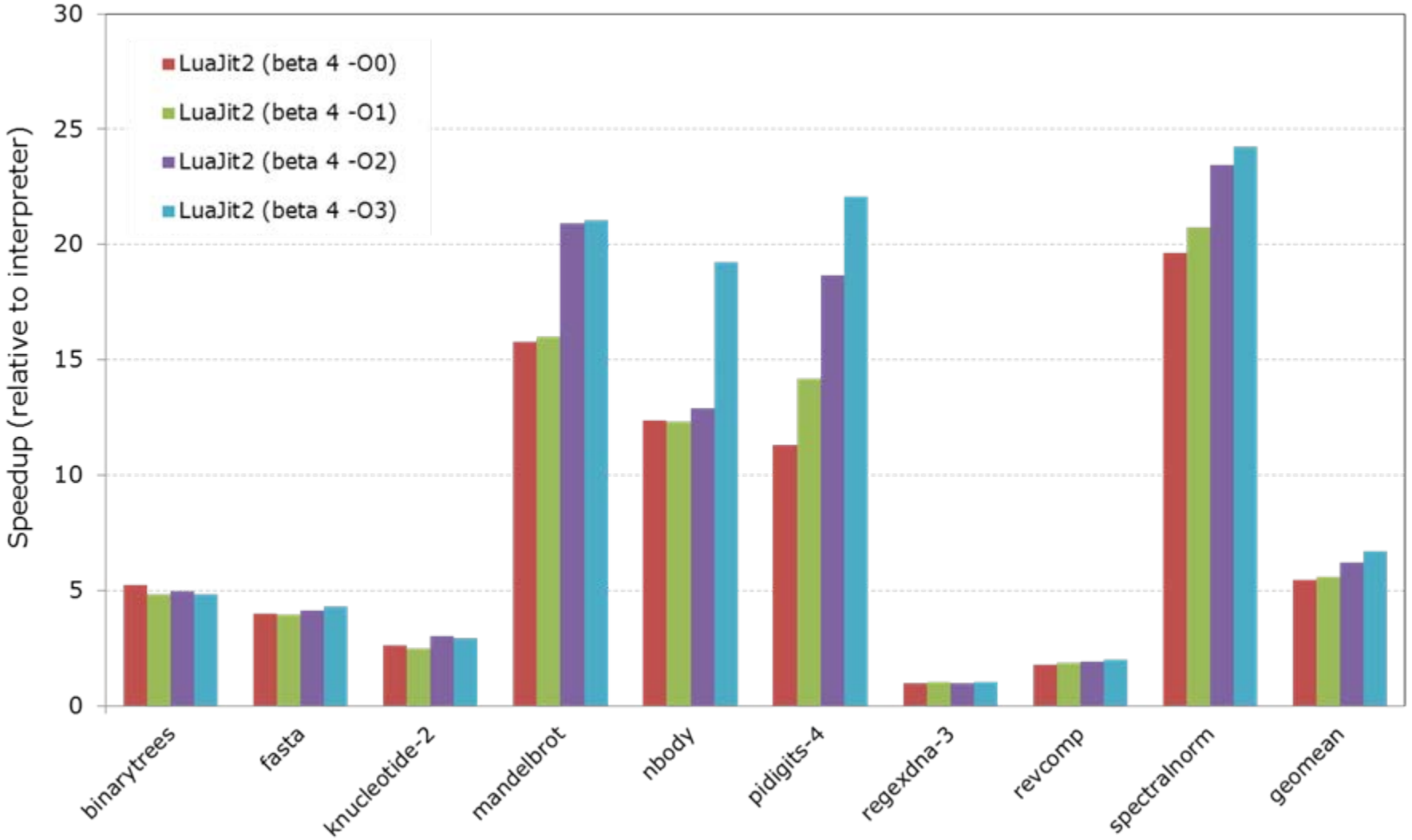
Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation

- BACK UP

© 2010 IBM Corporation

Performance of LuaJIT



Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation

PyPy (Interpreters + 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, support single inheritance
  - Numerical and string types use unboxed representations
  - Tuple, list, dictionary are homogeneous (across elements)
- Tracing JIT through both user program and runtime (RPython)
- Optimizations that work well
  - Removal of frame handling
  - Avoid creating temporary objects
  - Optimize attribute and name lookups

© 2010 IBM Corporation

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

© 2010 IBM Corporation

Jython

- Clean implementation of Python on top of JVM
  - Generate JVM bytecodes from Python 2.5 programs
    - interface with Java programs; cannot easily support standard C modules
  - Runtime is rewritten in Java, allow JIT optimize user programs and runtime
  - Python built-in objects are mapped to Java class hierarchy
    - allow (virtual) function specialization based on built-in types
- Large code explosion when applying standard JIT optimizations
- Large memory footprint
  - 300-600MB for small programs (~3MB on CPython)
- New InvokeDynamic bytecode in Java7 specification, but still not implemented in Jython

© 2010 IBM Corporation

Unladen-swallow

- Dealing with Dynamism
  - Caching LOAD_GLOBAL and import
  - Specialized binary and comparison operators, and builtin functions based on runtime feedback
  - Type inference for native types
- Implementation Improvements
  - Fast calls
  - Constantish
  - Expose Cpython stack to JIT (LLVM)
  - Omit untaken branches (during IR generation)

© 2010 IBM Corporation