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

Demographic evolution of programmers

- System programmers
- HPC
- CS programmers
- Domain experts
- Non-programmers

Application/programming

- New
- Traditional

Architecture

- Mixed workloads (data center)
- Big data workload (distributed)
- Streaming model (Hadoop, CUDA, OpenCL, SPL, …)
- Dynamic scripting Languages (javascript, python, php)
- SPEC, HPC, Database, Webserver
- C/C++, Fortran, Java, …
- Multi-core, general-purpose
- Accelerators (GPGPU, FPGA, SIMD)
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**

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

- **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

**Significant difference in JIT effectiveness across languages**
- Javascript has the most effective JITs
- Ruby JITs are similar to Python’s
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 facto standard implementation of Python)

```
foo.py

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

- **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

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

© 2012 IBM Corporation
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

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

```java
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

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

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

---

<table>
<thead>
<tr>
<th># Java bytecode</th>
<th>path length per Python loop iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) localvar-loop</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>heap-read</td>
<td>47</td>
</tr>
<tr>
<td>heap-write</td>
<td>11</td>
</tr>
<tr>
<td>heap-alloc</td>
<td>2</td>
</tr>
<tr>
<td>branch</td>
<td>46</td>
</tr>
<tr>
<td>invoke (JNI)</td>
<td>70(2)</td>
</tr>
<tr>
<td>return</td>
<td>70</td>
</tr>
<tr>
<td>arithmetic</td>
<td>18</td>
</tr>
<tr>
<td>local/const</td>
<td>268</td>
</tr>
<tr>
<td>Total</td>
<td>534</td>
</tr>
</tbody>
</table>

**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

Normalized Execution Time of Python JITs over CPython

Execution Time Normalized to CPython

Normalized Execution Time of Python JITs over CPython

- fiorano-hot
- unladen-swallow
- pypy_18

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

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

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

Execution Time Normalized to CPython

- django
- float
- nbody
- nqueens
- pystone
- richards
- rietveld
- slowpickle
- slowspitfire
- slowunpickle
- spambayes
- geomean

- noOpt
- cold
- warm
- hot

speedup

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

© 2012 IBM Corporation
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

<table>
<thead>
<tr>
<th># Java Bytecode</th>
<th>Instruction path length per python bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD_LOCAL</td>
<td>BINARY_ADD (int+int)</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>heap-read</td>
<td>3</td>
</tr>
<tr>
<td>heap-write</td>
<td>0</td>
</tr>
<tr>
<td>heap-alloc</td>
<td>0</td>
</tr>
<tr>
<td>branch</td>
<td>2</td>
</tr>
<tr>
<td>invoke (JNI)</td>
<td>0</td>
</tr>
<tr>
<td>return</td>
<td>0</td>
</tr>
<tr>
<td>arithmetic</td>
<td>0</td>
</tr>
<tr>
<td>local/const</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>12</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Idiom</th>
<th>pypy_18</th>
<th>fiorano</th>
<th>jython 2.5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLS</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>136x</td>
<td>98x</td>
<td>23x</td>
</tr>
<tr>
<td>ARITHMETIC</td>
<td>23x</td>
<td>35x</td>
<td>40x</td>
</tr>
<tr>
<td>NEW_INSTANCE</td>
<td>136x</td>
<td>98x</td>
<td>23x</td>
</tr>
<tr>
<td>CONTROL_FLOW</td>
<td>136x</td>
<td>98x</td>
<td>23x</td>
</tr>
<tr>
<td>STRING</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
<tr>
<td>UNICODE</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
<tr>
<td>DICTIONARY</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
<tr>
<td>LIST</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
<tr>
<td>TUPLES</td>
<td>37x</td>
<td>122x</td>
<td>29x</td>
</tr>
</tbody>
</table>
Breakdown of Dynamic Python Bytecode Execution

- Interpreted
- Compiled-unspecializable
- Compiled-specialization-succeeded
- Interpreted-guard-failed
- Compiled-unspecialized
- Compiled-specialization-failed
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
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

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

Safari Performance

Surf at the speed of fast.

As you surf the web, your fingers will feel the responsiveness of the new Nitro JavaScript engine powering Safari. It runs JavaScript up to twice as fast as in iOS 4.2.

- Accelerated evolution in the last few years
- Highly competitive

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

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

![Bar chart showing the speedup of different Javascript implementations.](image)

- **TraceMonkey**
- **V8**
- **Rhino**

- **binarytrees**: Speedup of 56 (TraceMonkey), 19 (V8), 4 (Rhino)
- **fasta**: Speedup of 2 (TraceMonkey), 2 (V8), 4 (Rhino)
- **mandelbrot**: Speedup of 8 (TraceMonkey), 16 (V8), 8 (Rhino)
- **nbody**: Speedup of 2 (TraceMonkey), 2 (V8), 4 (Rhino)
- **spectralnorm**: Speedup of 2 (TraceMonkey), 2 (V8), 4 (Rhino)
- **geomean**: Speedup of 2 (TraceMonkey), 2 (V8), 4 (Rhino)

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

© 2012 IBM Corporation
Performance of Ruby Implementations

![Chart showing speedup of various Ruby implementations compared to Ruby 1.9.](image)

- Ruby 1.9
- Jruby/OpenJDK
- Jruby/Hotspot 7
- Jruby/TR
- Rubinius

The chart illustrates the speedup of various Ruby implementations compared to Ruby 1.9.
IronPython: DynamicSites

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

```csharp
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