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

Kazuaki Ishizaki [+], Takeshi Ogasawara [+], Jose Castanos [*], Priya Nagpurkar [*], David Edelsohn [*], Toshio Nakatani [+]

[+]IBM Research – Tokyo
[*]IBM T.J. Watson Research Center

Improve performance of a dynamically-typed language by reusing an existing JIT compiler

- Dynamically-typed languages are becoming popular
  - Perl, PHP, JavaScript, Python, Ruby, Lua, …
  - Examples of large applications
    - Hulu (Ruby), Washington post (Python)
- Performance is an issue compared to statically-typed languages
  - Python, PHP, and Ruby are 2.2~6.5x slower than Java (interpreter only)
    [Computer Language Benchmarks Game 2009]
- Developing a JIT compiler for each language from scratch is too costly
  - There are matured JIT compilers for a statically-typed language

Performance overheads in dynamically-typed language

- Every variable can be dynamically-typed
  - Need type checks
- Every statement can potentially throw exceptions due to type mismatch and so on
  - Need exception checks
- Every field and symbol can be added, deleted, and changed at runtime
  - Need access checks
- A type of every object and its class hierarchy can be changed at runtime
  - Need class hierarchy checks

```
a = obj.x + 1.2
if (isinstance(a, Integer)):
    …
```

Our Contributions

- Reduce performance overheads in dynamically-typed language
  - By compiler optimizations
    - Exception checks
  - By optimized runtime
    - Type checks
    - Access checks
    - Class hierarchy checks
- Evaluate performance improvement by each optimization
  - Our JIT compiler improves performance by 1.76x against Python language interpreter

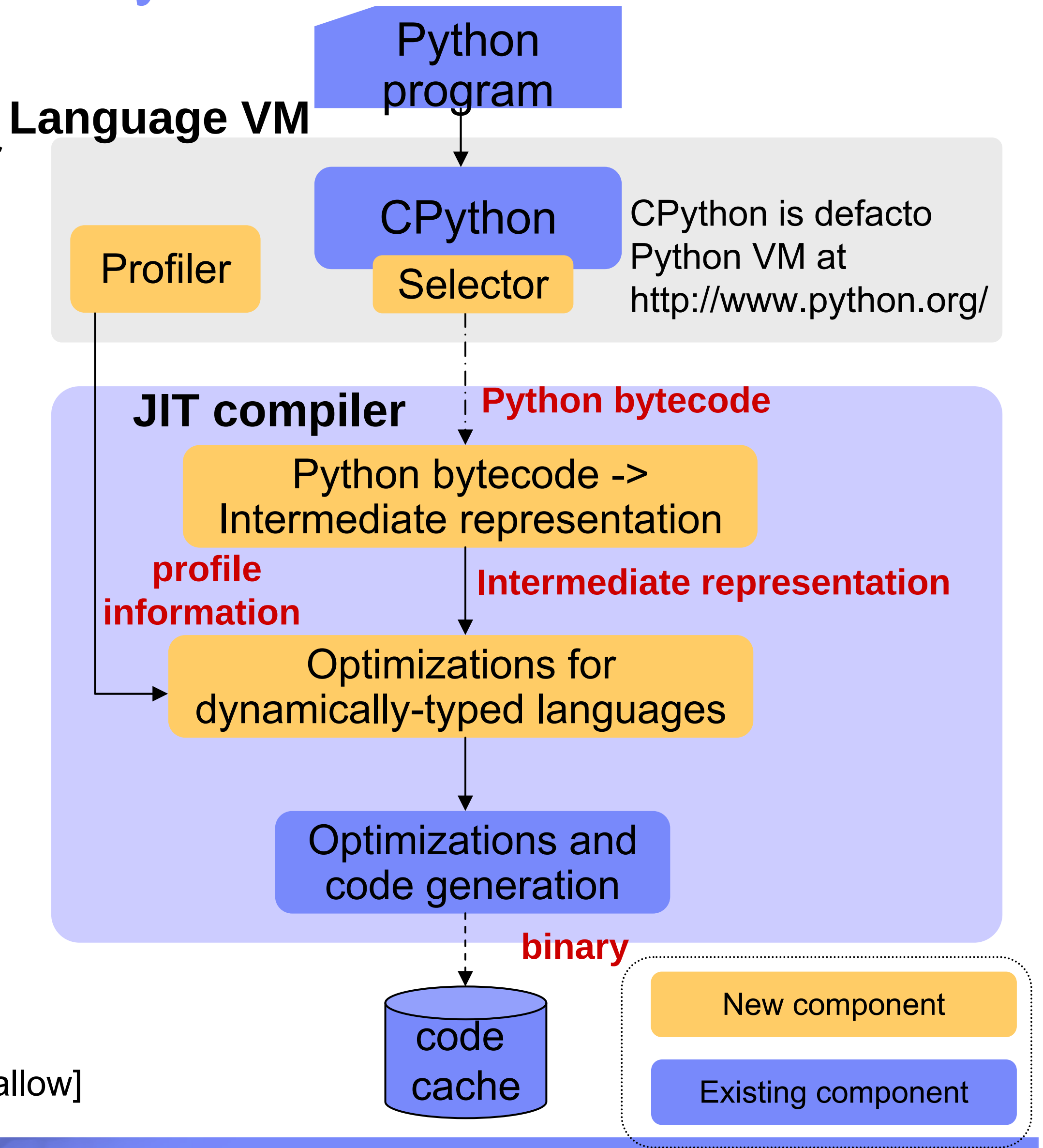
Outline

- Motivation & Goal
- Contributions
- Overview of our Approach
- Our Optimizations
- Performance Evaluation
- Related Works
- Conclusion and Future Work

High level overview of our Python runtime

- IBM production-quality Just-In-Time (JIT) compiler for Java as a base
- CPython as a language virtual machine (VM)
  - Maintain compatibility with existing libraries coupled with CPython
    - E.g. mod_wsgi for using apache web server

➢ Same structure as Unladen Swallow
  - CPython with LLVM compiler infrastructure [http://code.google.com/p/unladen-swallow]



Optimizations evaluated for performance

Optimization	Source of overhead	Novelty compared to Unladen Swallow
Reduce overhead to look up a hash when access a field	Dynamically-typed	New
Reduce overhead to check a given object is an instance of a class	Dynamically-typed	New
Reduce overhead to search a dictionary when call hasattr()	Dynamically-typed	New
Map operand stack to stack-allocated variables	Python	New
Represent a exception check without splitting a basic block	Dynamically-typed	New
Specialization for one operation using runtime type information	Dynamically-typed	Improvement
Speculatively constantish global variables and built-in functions	Dynamically-typed	Improvement
Represent an operation to maintain reference counting without branch	Python	Same
Map Python’s local variable to stack-allocated variables	Python	Same

➔ See paper for details of each optimization

Statically-typed language v.s. dynamically-typed language

Statically-typed language

```
S1: Number a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

Dynamically-typed language

```
S1: a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

Comparison of an access to a field

Statically-typed language

```
S1: Number a = obj.x
S2: if (isinstance(a, Integer)):
      …
```

a = load offset_for_field#x[obj]

- Get a value the field x by accessing with constant offset

One instruction

Dynamically-typed language

```
S1: a = obj.x
S2: if (isinstance(a, Integer)):
      …
```

a = call get_value_dict(obj, field#x)

hash

	field name	*value*
0	z	5
…	…	…
8	y	[1, 2]
9	x	3.4

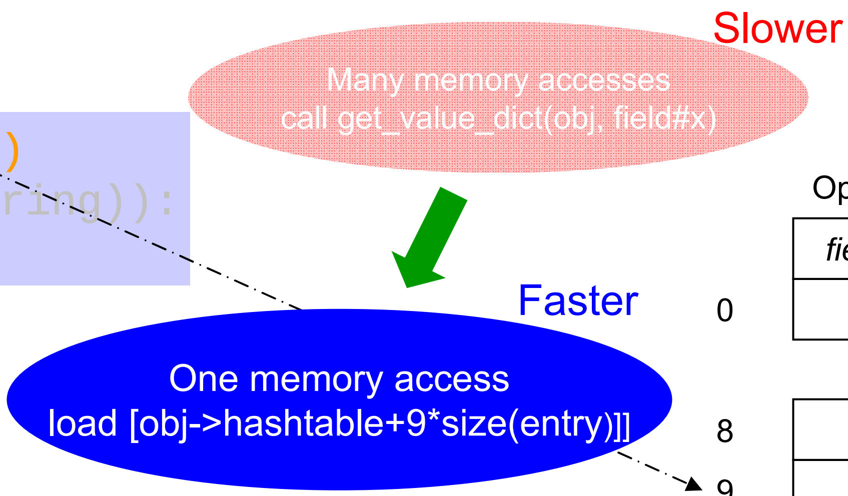
- Get a value the field x by looking up a hash with conflict resolution using many memory accesses

10~ instructions

Access to a field without conflict resolution

- Access an value using profiled index without conflict resolution when look up hash
  - Profile an offset of open-addressed hash table at S1 *before compilation*
    - Profiled index = 9 for the field name x
  - Generate code to access an entry at index = 9 in the table *at compilation time*
  - Access an entry (index = 9) with validation check *at runtime*

```
S1: a = obj.x (index = 9)
S2: if (isinstance(a, String)):
        ….
```



Open-addressed hash table

	field name	*value*
0	z	5
⋮		
8	y	[1, 2]
9	x	3.4

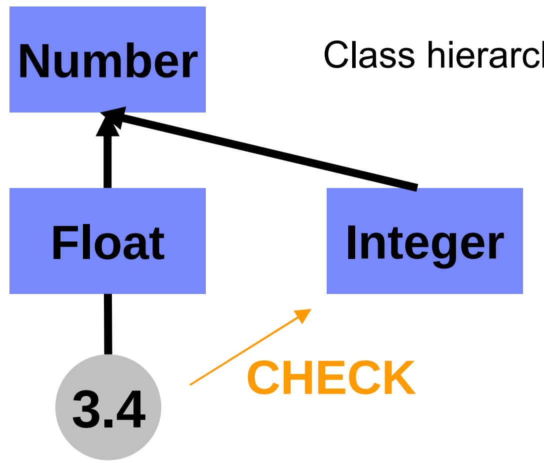
The result of dynamically-typed language can vary for the same instance check

Statically-typed language

```
S1: Number a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

Always False for a=3.4

Class hierarchy

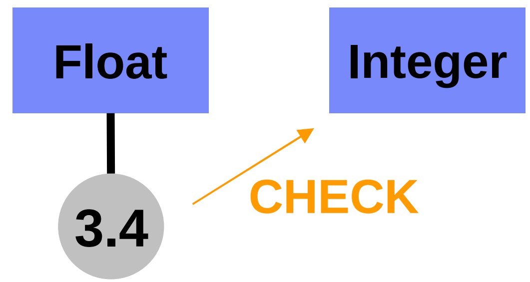


Dynamically-typed language

```
S1: a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

False for a=3.4

Class hierarchy



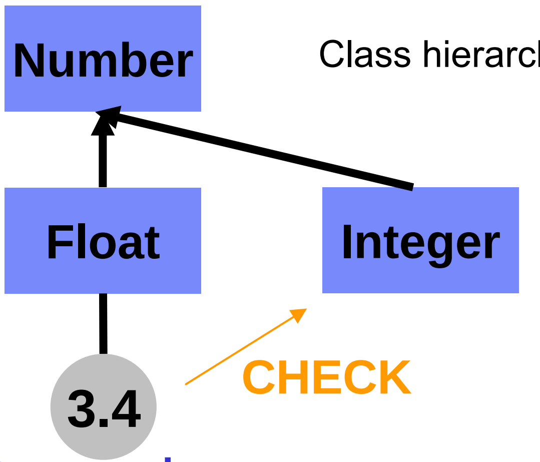
The result of dynamically-typed language can vary for the same instance check

Statically-typed language

```
S1: Number a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

Always False for a=3.4

Class hierarchy



Naïve cache or pre-computation is effective

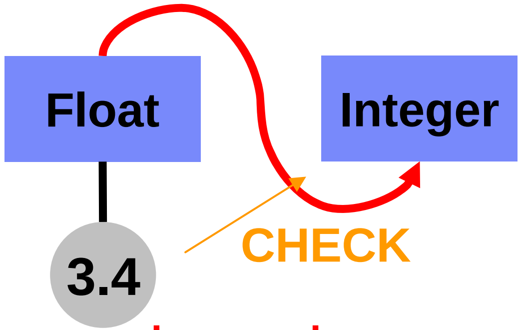
Dynamically-typed language

```
S1: a = obj.x
S2: if (isinstance(a, Integer)):
    …
```

True for a=3.4 after class hierarchy change at runtime

Class hierarchy

Float becomes a subclass of Integer



Naïve cache and pre-computation cannot be applicable

Caching the results of instance checks

```
S1: a = obj.x
S2: if (isinstance(a, Integer)):
        …
```

- Our JIT compiler already had a component for *Java* for caching frequently-checked classes of target objects and the results of the checks.

```
r1 = a→class
cmp r1, freqClass          // profiled class for a
jne slow_instance_check
r2 = cachedResult          // result by comparing freqClass
                              with Integer
```

- We extended this component for Python.
  - Add the code for validation of the reusability of cached results

Performance evaluation

- Measured performance improvement by each optimization or set of optimizations
  - at steady state performance
  - by disabling each optimization or a set of optimizations

- Hardware & OS
  - 2.93-GHz Intel Xeon X5670 (disabled turbo boost) with 24-GB memory
  - Redhat Linux 5.5
- Our runtime for Python
  - CPython 2.6.4 (32bit) with IBM production-quality JIT compiler
- Benchmarks
  - Unladen Swallow benchmark suite
    [http://code.google.com/p/unladen-swallow/wiki/Benchmarks]

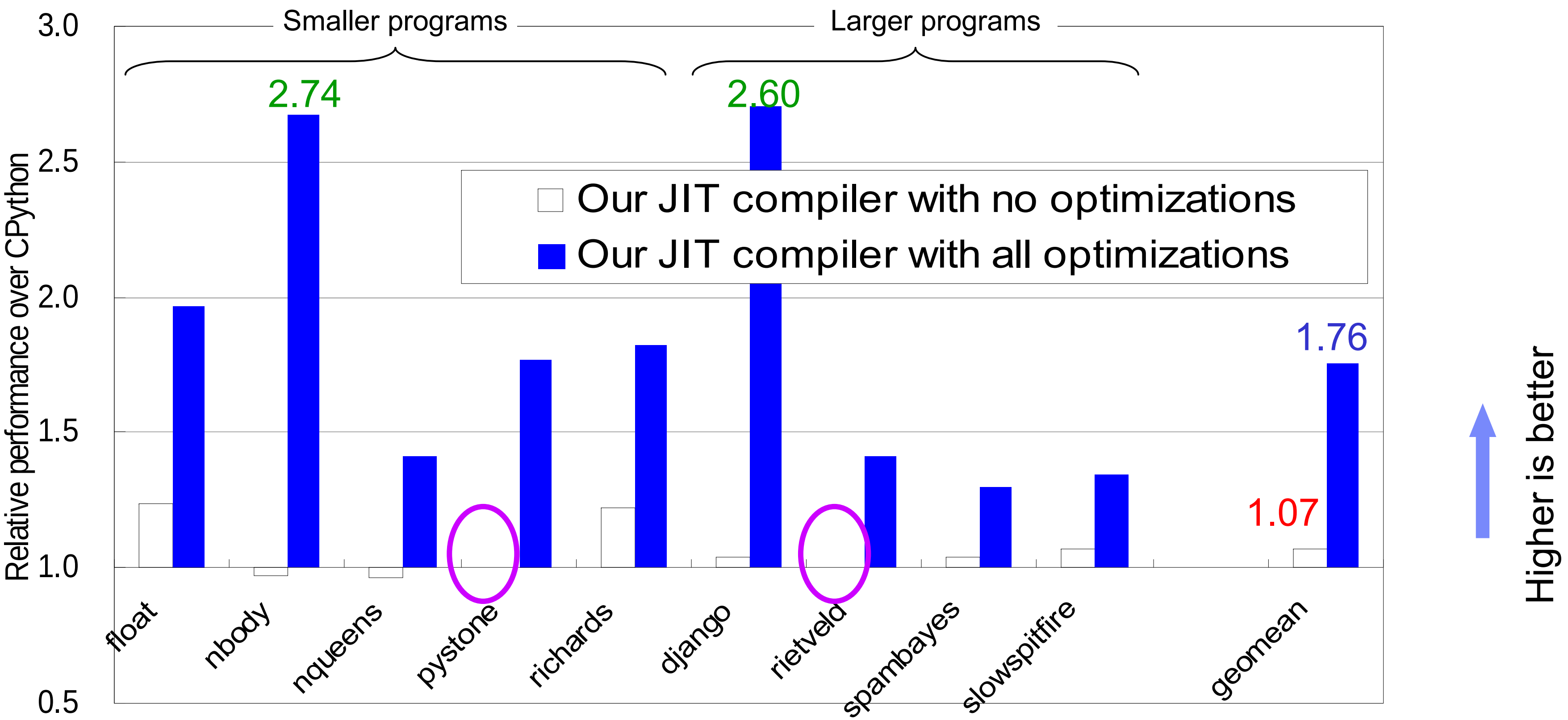
Optimizations evaluated for performance

Optimization	Source of overhead	Novelty compared to Unladen Swallow
Reduce overhead to look up a hash when access a field	Dynamically-typed	New
Reduce overhead to check a given object is an instance of a class	Dynamically-typed	New
Reduce overhead to search a dictionary when call hasattr()	Dynamically-typed	New
Map operand stack to stack-allocated variables	Python	New
Represent a exception check without splitting a basic block	Dynamically-typed	New
Specialization for one operation using runtime type information	Dynamically-typed	Improvement
Speculatively constantish global variables and built-in functions	Dynamically-typed	Improvement
Represent an operation to maintain reference counting without branch	Python	Same
Map Python's local variable to stack-allocated variables	Python	Same

➔See paper for details of each optimization

Our JIT compiler improves by 1.76x against CPython interpreter

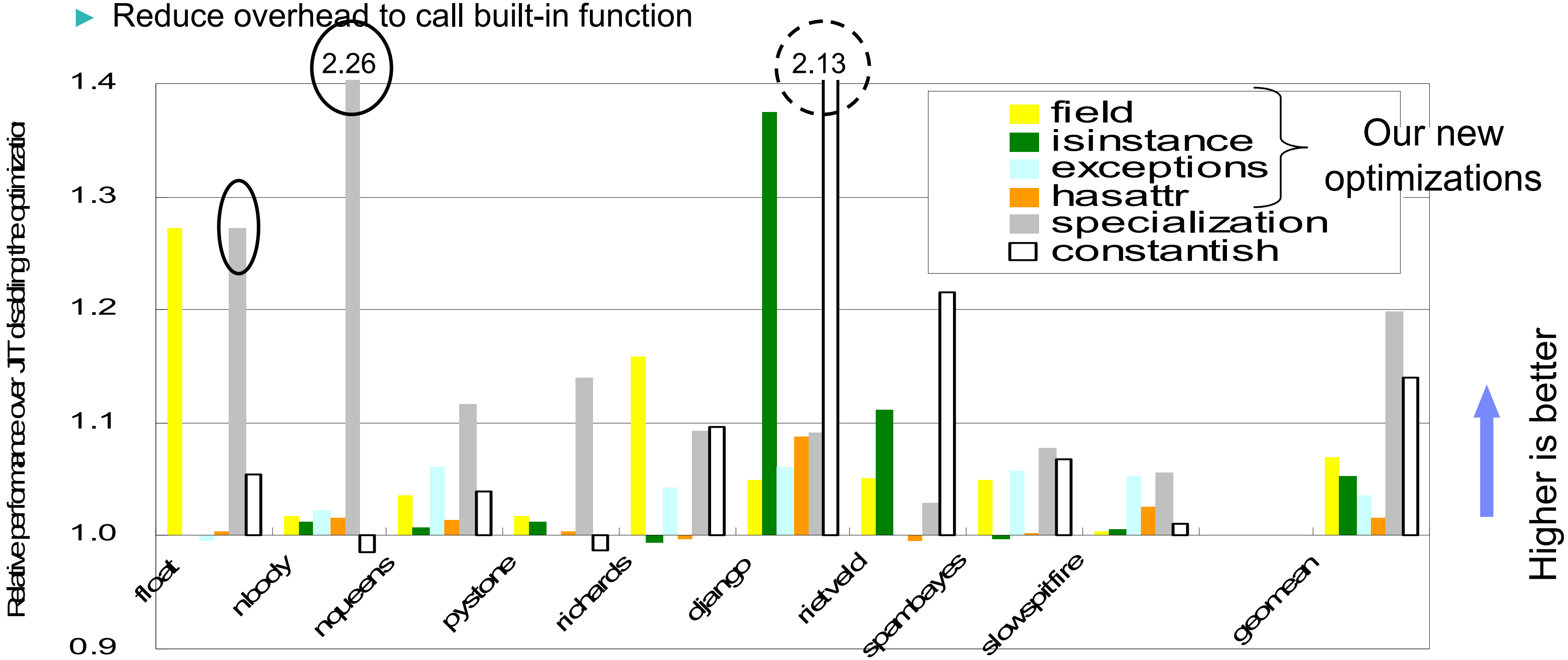
- nbody is 2.74x faster and django is 2.60x faster
- Our JIT w/o all of optimizations for dynamically-typed languages is 1.07x faster than CPython interpreter
- pystone and rietveld fail due to overflow of compiler working memory



Performance improvements by reducing overhead in dynamically-typed language

- "field" are effective for float and richards
- "isinstance" is effective for django and rietveld
  - Django framework uses many instance checks
- "specialization" is effective for float and nbody
- "constantish" is effective for django
  - Reduce overhead to call built-in function

Our new optimizations



Related work

- Untouch JIT compiler for Java bytecode or common intermediate language (CIL)
  - Jython[http://jython.org//], IronPython[http://ironpython.net/] (Python)
  - Jruby[http://jruby.org/], IronRuby[http://ironruby.net/] (Ruby)
  - Rhino[http://www.mozilla.org/rhino/] (Javascript)
- Enhance JIT compiler
  - Unladen swallow[http://code.google.com/p/unladen-swallow/] (Python with LLVM[Lattner2004])
  - Rubinius[http://rubini.us/] (Ruby with LLVM)
- Create JIT compiler and runtime from scratch
  - V8[http://code.google.com/p/v8/], TraceMonkey[Gas2009], SpiderMonkey[http://www.mozilla.org/js/spidermonkey/], … (Javascript)
  - PyPy[Boltz2011] (Python)

Summary of Our Accomplishment

Reducing performance overhead
in dynamically-typed language
by enhancing JIT compiler for Java

- Future work
  - Apply aggressive compiler optimizations for a dynamically-typed language
    - Implementing type specialization within a method
    - Implementing unboxing for primitive types : int and float
  - Exploit existing compiler optimizations furthermore
    - e.g. common subexpression elimination for accessing a field and type flow optimization