Unladen Swallow: Fewer coconuts, faster Python

Collin Winter, Jeffrey Yasskin
{collinwinter,jyasskin}@google.com

unladen-swallow@googlegroups.com
#unladenswallow on OFTC
Why did Google start this?

- Lots of Python at Google.
  - Python one of Google’s three primary languages.
  - Python enables fast development, rapid prototyping.
  - Engineers should use the language they want...
  - ...and it should be fast.

- Biggest user: YouTube
  - YouTube is pure-Python.
  - #2 search site on the Internet, behind google.com.
Goals

- Make Python 5x faster.
- Source-compatible with existing Python code.
- Source-compatible with existing C extension modules.
- Focus on ease of migration.
- Open-source everything, merge back into CPython.

Baseline requirements:
- Embeddable in C++ applications.
- Support existing C extension modules and SWIG.
- Compatible with all our existing applications.
- Compatible with our existing infrastructure.
- As fast, or faster, than CPython.
- Baseline: branch CPython.
Why is Python slow?

```python
def add(a, b):
    return a + b
```

- Everything is an object.
- Everything is a method call, eventually.
- Ducktyping means we can’t statically predict receiver types.
def add(a, b):
    return a + b
Why is Python slow?

def foo(x):
    yield len(x)
    yield len(x)

>>> g = foo(range(5))
>>> g.next()
5
>>> len = lambda y: 8
>>> g.next()
8
How do you make Python faster?

- CPython today:
  - Stack-based bytecode interpreter.
  - Missed the last 30 years of research.

- Conservative ideas:
  - Computed goto-based interpreter loop.
  - Add new, specialized opcodes.
  - Superinstructions (WPython)
Unladen Swallow

- Unladen Swallow: just-in-time compilation.
  - Preserve C extension compatibility.
  - Use LLVM for code generation, optimization.
  - Use Clang for inlining C functions.
  - Runtime feedback for specialization.
  - Inspired by Self-93, HotSpot, V8, Psyco.
Top-down Inlining Opportunities

```python
>>> x = dict()
>>> len(x)
```

...len() calls PyObject_Size()
...which looks up x->ob_type->tp_as_sequence->sq_length
...which is NULL, so call PyMapping_Size()
...which looks up x->ob_type->tp_as_mapping->mp_length
...which is a function pointer to dict_length()
...which returns x->ma_used
Using Clang and llc for Inlining

- Pipeline: C --> LLVM .bc --> C++ API calls
- Inline useful macros:

```c
/*** Python/llvm_inline_functions.c ****/

void __attribute__((always_inline))
  _PyLlvm_WrapDecref(PyObject *obj)
{
    Py_DECREF(obj);
}

- Uses a custom single-function inlining pass.
Python Objects: constant(ish)

```c
struct PyDictObject {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
    Py_ssize_t ma_fill;
    Py_ssize_t ma_used;
    ...
};

PyTypeObject PyDict_Type = {
    ...
    &dict_as_sequence,
    &dict_as_mapping,
    ...
};
```
Experience with LLVM

• Generally very positive!

• Obstacles overcome:
  • JIT bugs.
  • gdb/oprofile support.
  • Some quadratic behaviour.

• Obstacles remaining:
  • LLVM does not cure cancer.
  • Some optimizations missing.
  • More lurking quadratic behaviour.
  • Python semantics.

• Stop; collaborate; listen.
Measurement & Testing

- Benchmarks representing real-world applications:
  - YouTube hotspots: Spitfire templates.
  - Libraries: pickling, regular expressions.
  - Apps: 2to3, Django.
  - Microbenchmarks: GC, IO, string operations.

- Correctness:
  - SWIGed code, extensions used by YouTube.
  - Google’s large Python codebase.
  - Large Python projects: Mercurial, NumPy, Twisted, etc.
  - Randomized testing.
Status Report

- Two releases so far:
  - 2009Q1: 15-20% faster than CPython.
    - cPickle 2x faster.
  - 2009Q2: 10% faster than Q1; full JIT on top of LLVM.

- 2009Q3: stabilizing, close to release.
  - Optimized LOAD_GLOBAL opcode.
  - Optimized calls to C functions.
  - Less unnecessary error checking.
  - Better constant propagation.
  - More data exposed to LLVM
  - gdb, oprofile support.
  - 20-75% faster than Q2.
Looking Forward: Q4

• 2009Q4:
  • More typefeed back!
  • More inlining!
  • More Clang compilation!
  • Fewer frame allocations!
  • Fewer GIL checks!
  • Begin merger with CPython!
GIL?
Questions?

http://code.google.com/p/unladen-swallow/

collinwinter@google.com
unladen-swallow@googlegroups.com
#unladenswallow on OFTC
Backup Slides
def foo(x):
    ...
    y = len(x)
    ...

    ... LOAD_GLOBAL 7 (len)
    LOAD_FAST 0 (x)
    CALL_FUNCTION 1
    STORE_FAST 4 (y)
    ...

Make it faster: specialize dynamically
... LOAD_GLOBAL 7 (len) ...

x = PyDict_GetItem(globals, "len")
if (x == NULL) {
    x = PyDict_GetItem(builtins, "len")
    if (x == NULL) {
        PyErr_NoGlobals()
        return -1;
    }
}

Make it faster: LOAD_GLOBAL

... LOAD_GLOBAL 7 (len) ...

if (world_has_not_changed) {
    x = (PyObject *)11712432
}
else {
    goto bail_to_interpreter
}
Make it faster: if (world_has_not_changed) {

def foo(x):
    ...
    y = len(x)
    ...

def bar(x):
    globals len
    len = ...

def template(data, rows):
    for row in rows:
        data.append("<tr>")
        for col in row:
            data.append("<td>%s</td>" % col)
        data.append("</tr>")

>>> our_data = []
>>> template(our_data, our_rows)
>>> print "".join(our_data)
Make it faster: call sites

```
data.append("<td>%s</td>" % col)
```

- LOAD_FAST 0 (data)
- LOAD_ATTR 0 (append)
- LOAD_CONST 1 ('<td>%s</td>')
- LOAD_FAST 1 (col)
- BINARY_MODULO
- CALL_FUNCTION 1
LOAD_FAST 0 (data)
LOAD_ATTR 0 (append)
...
CALL_FUNCTION 1

data = locals[0];
if (Py_TYPE(data) != EXPECTED_TYPE)
    goto bail_to_interpreter;
if (Py_TYPE_VERSION(data) != EXPECTED_VERSION)
    goto_bail_to_interpreter;
// LOAD_CONST
// LOAD_FAST
// BINARY_MODULO
retval = list_append(data, modulo_result);