Loop-Aware Optimizations in PyPy's Tracing JIT

Håkan Ardö[1] Carl Friedrich Bolz[2] Maciej Fijałkowski

[1]Centre for Mathematical Sciences, Lund University

[2]Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany

2012 DLS, 22nd of October, 2012

RPython and PyPy

- Context: RPython
- a language for writing interpreters for dynamic languages
- a generic tracing JIT, applicable to many languages
- used to implement PyPy, an efficient Python interpreter

Tracing JITs Compile by Observing an Interpreter

- VM contains both an interpreter and the tracing JIT compiler
- JIT works by observing and logging what the interpreter does
- for interesting, commonly executed code paths
- produces a linear list of operations (trace)

Why do tracing JITs work?

- They are good at selecting interesting and common code paths
- both through the user program and through the runtime
- the latter is particularly important for dynamic languages with a big runtime like Python

Why do tracing JITs work?

- They are good at selecting interesting and common code paths
- both through the user program and through the runtime
- the latter is particularly important for dynamic languages with a big runtime like Python
- traces are trivial to optimize

Optimizing traces

- A trace is an extended basic block
- (it has one entry and several exits)
- traces are easy to optime due to lack of control flow merges

Optimizing traces

- A trace is an extended basic block
- (it has one entry and several exits)
- traces are easy to optime due to lack of control flow merges
- most optimizations are one forward pass
- optimizers are often like symbolic executors
- can do optimizations that are expensive or even untractable with full control flow

Example

$$
\begin{aligned}
w_0 &= a_0 + 0 \\
x_0 &= w_0 + 1 \\
y_0 &= a_0 + 1 \\
z_0 &= x_0 + y_0 \\
&\ldots \\
&\mathrm{jump}(L_0, z_0, b_0)
\end{aligned}
$$

Example

$\cancel{w_0 = a_0 + 0}$
$x_0 = w_0 + 1$
$y_0 = a_0 + 1$
$z_0 = x_0 + y_0$
$\ldots$
$\text{jump}(L_0, z_0, b_0)$

rename w_0 to a_0

Example

$\cancel{w_0 = a_0 + 0}$
$x_0 = a_0 + 1$
$y_0 = a_0 + 1$
$z_0 = x_0 + y_0$
$\ldots$
$\mathrm{jump}(L_0, z_0, b_0)$

rename w_0 to a_0
$a_0 + 1$ in x_0

Example

$\cancel{w_0 = a_0 + 0}$

$x_0 = a_0 + 1$

$\cancel{y_0 = a_0 + 1}$

$z_0 = x_0 + y_0$

$\ldots$

$\text{jump}(L_0, z_0, b_0)$

rename w_0 to a_0

$a_0 + 1$ in x_0

rename y_0 to x_0

Example

$\cancel{w_0 = a_0 + 0}$

$x_0 = a_0 + 1$

$\cancel{y_0 = a_0 + 1}$

$z_0 = x_0 + x_0$

$\ldots$

$\mathrm{jump}(L_0, z_0, b_0)$

rename w_0 to a_0

$a_0 + 1$ in x_0

rename y_0 to x_0

$x_0 + x_0$ in z_0

Problems with this approach

- most traces actually are loops
- naive foward passes ignore this bit of control flow optimization available
- how to fix that without sacrifing simplicity of optimizations?

Idea for solution

- idea first proposed and implemented in LuaJIT by Mike Pall
- this talk presents the implementation of the same approach in RPython's tracing JIT

Idea for solution

- idea first proposed and implemented in LuaJIT by Mike Pall
- this talk presents the implementation of the same approach in RPython's tracing JIT

Approach

- do a pre-processing step on the traces
- apply the unchanged forward-pass optimizations
- do some post-processing
- pre-processing is done in such a way that the normal optimizations become loop-aware

Idea for solution

- idea first proposed and implemented in LuaJIT by Mike Pall
- this talk presents the implementation of the same approach in RPython's tracing JIT

Approach

- do a pre-processing step on the traces
- apply the unchanged forward-pass optimizations
- do some post-processing
- pre-processing is done in such a way that the normal optimizations become loop-aware
- intuition: give the optimizations a second iteration of context to work with

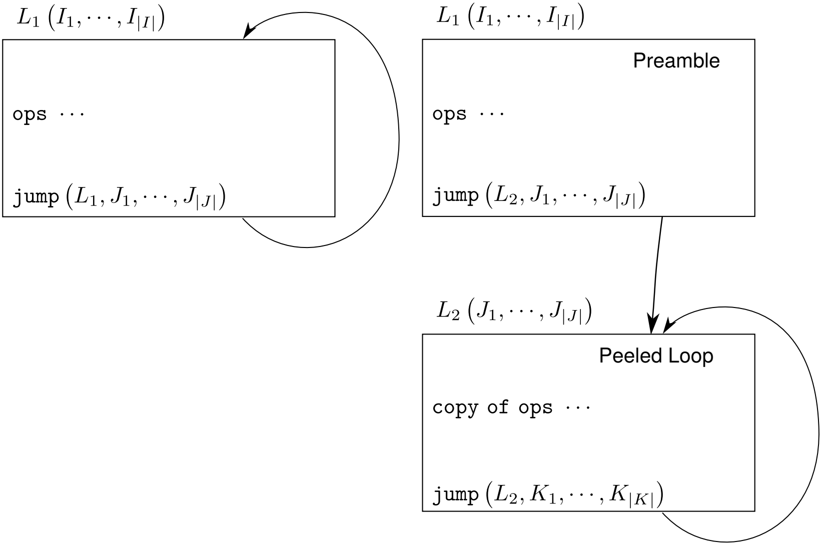
Pre-processing the loops

- pre-processing does loop unrolling
- peels off one iteration of the loop, duplicating the trace
- the optimizations optimize both iterations together
- this yields loop-invariant code motion and related optimizations

Loop Peeling

Original Loop:

After Loop Peeling:



Loop Peeling

```
L_0(i_0):
i_1 = i_0 + 1
print(i_1)
jump(L_0, i_0)
```

Loop Peeling

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₀, i₀)
```

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₁, i₀)

L₁(i₀):
i₂ = i₀ + 1
print(i₂)
jump(L₁, i₀)
```

Apply Optimizations

```
L_0(i_0):
i_1 = i_0 + 1
print(i_1)
jump(L_1, i_0)

L_1(i_0):
i_2 = i_0 + 1
print(i_2)
jump(L_1, i_0)
```

Apply Optimizations

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₁, i₀)

L₁(i₀):
i₂ = i₀ + 1
print(i₂)
jump(L₁, i₀)
```

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₁, i₀)

L₁(i₀):
print(i₁)
jump(L₁, i₀)
```

Add extra arguments

```
L_0(i_0):
i_1 = i_0 + 1
print(i_1)
jump(L_1, i_0)

L_1(i_0):
print(i_1)
jump(L_1, i_0)
```

Add extra arguments

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₁, i₀)

L₁(i₀):
print(i₁)
jump(L₁, i₀)
```

```
L₀(i₀):
i₁ = i₀ + 1
print(i₁)
jump(L₁, i₀, i₁)

L₁(i₀, i₁):
print(i₁)
jump(L₁, i₀, i₁)
```

Optimizations helped by loop peeling

- redundant guard removal
- common subexpression elimination
- heap optimizations
- allocation removal

Larger example

```
while True:
    y = y + 1
```

Larger example

```
while True:
    y = y + 1
```

$L_0(p_0, p_1)$:
guard_class(p_1, BoxedInteger)
i_2 = get(p_1, intval)
guard_class(p_0, BoxedInteger)
i_3 = get(p_0, intval)
$i_4 = i_2 + i_3$
p_5 = new(BoxedInteger)
set(p_5, intval, i_4)
jump(L_0, p_0, p_5)

Peeled trace

```
L0(p0, p1):
guard_class(p1, BoxedInteger)
i2 = get(p1, intval)
guard_class(p0, BoxedInteger)
i3 = get(p0, intval)
i4 = i2 + i3
p5 = new(BoxedInteger)
set(p5, intval, i4)
jump(L1, p0, p5)

L1(p0, p5):
guard_class(p5, BoxedInteger)
i6 = get(p5, intval)
guard_class(p0, BoxedInteger)
i7 = get(p0, intval)
i8 = i6 + i7
p9 = new(BoxedInteger)
set(p9, intval, i8)
jump(L1, p0, p9)
```

Final trace

```
L_0(p_0, p_1):
guard_class(p_1, BoxedInteger)
i_2 = get(p_1, intval)
guard_class(p_0, BoxedInteger)
i_3 = get(p_0, intval)
i_4 = i_2 + i_3
jump(L_1, p_0, i_4)

L_1(p_0, i_3, i_4):
i_8 = i_4 + i_3
jump(L_1, p_0, i_3, i_8)
```

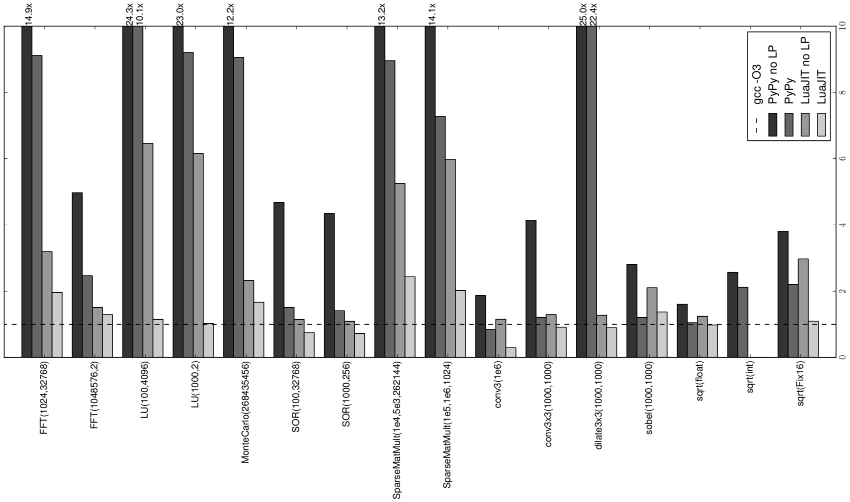
Results

- a number of numeric kernels
- some for image processing
- some from SciMark
- comparison against GCC and LuaJIT

Results

- a number of numeric kernels
- some for image processing
- some from SciMark
- comparison against GCC and LuaJIT
- geometric mean of speedups of loop peeling is 70%

Benchmark Results



Conclusion

- a simple preprocessing step on traces enables loop-aware optimizations for tracing JITs
- only minimal changes to the existing optimizations necessary

Demo

- Video analytics research example
- Experimenten driven - prototyping
- Custom loops over the pixels
- Good enough performace

- Image class with task specific features
  - Zero-padded
  - Clips updates outside border
- @autoreload decorator reloading functions on code change
- ReloadHack class reloads and reinstanciates on code change

Image class

```
class Image(object):
    def __getitem__(self, (x, y)):
        if 0 <= x < self.width and 0 <= y < self.height:
            return self.data[y * self.width + x]
        return 0

    def __setitem__(self, (x, y), value):
        if 0 <= x < self.width and 0 <= y < self.height:
            self.data[y * self.width + x] = value

    __add__ = binop(float.__add__)
    __sub__ = binop(float.__sub__)
    __mul__ = binop(float.__mul__)
    __div__ = binop(float.__div__)
    __pow__ = binop(float.__pow__)

   ...
```

Image class

```python
def binop(op):
    def f(a, b):
        if not isinstance(a, Image):
            a = ConstantImage(b.width, b.height, a)
        if not isinstance(b, Image):
            b = ConstantImage(a.width, a.height, b)

        out = a.new(typecode='d')
        for x, y in a.indexes():
            out[x, y] = op(float(a[x, y]), float(b[x, y]))

        return out
    return f
```