Loop-Aware Optimizations in PyPy's Tracing JIT

Håkan Ardö Carl Friedrich Bolz Macieij Fijałkowski

Vincent Foley-Bourgon
COMP-763 - Fall 2013
McGill University

September 2013

Plan

1. About the paper
2. The Big Idea
3. Running example
4. Details
5. Benchmarks

About the paper

About the paper

- Håkan Ardö, Carl Friedrich Bolz, Macieij Fijałkowski
- Published in 2012
- Presented at DLS '12, in Tucson, AZ
- Report on the implementation of a LuaJIT optimization technique in PyPy

The Big Idea

The Big Idea

Tracing JIT

1. Code is ran in an interpreter
2. Loop iterations are counted; if they exceed a certain threshold, they are considered hot
3. Hot loop is traced; executed instructions are recorded in a trace[1]
4. Trace is compiled to native code
5. Native code is executed in subsequent iterations
6. Speed-up!

[1] With guards to ensure conditions still hold

The Big Idea

Trivial example

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

- Infinite loop
- i_0 is loop-invariant
- Always compute the same value $i_1 = i_0 + 1$
- Always print the same value

The Big Idea

Loop-invariant code motion

We'd like to move the addition outside the loop body.

BUT

The trace doesn't contain the code before the loop body.
Also, we'd like to reuse existing optimizations.

The Big Idea

Loop peeling (step 1)

We duplicate the loop body.

```
1   # Preamble
2   L_0(i_0):
3   i_1 = i_0 + 1
4   print(i_1)
5   jump(L_0, i_0)
6
7   # Peeled loop
8   L_0(i_0):
9   i_1 = i_0 + 1
10  print(i_1)
11  jump(L_0, i_0)
```

The Big Idea

Loop peeling (step 2)

We update the label names.

```
1   # Preamble
2   L_0(i_0):
3   i_1 = i_0 + 1
4   print(i_1)
5   jump(L_1, i_0)
6
7   # Peeled loop
8   L_1(i_0):
9   i_1 = i_0 + 1
10  print(i_1)
11  jump(L_1, i_0)
```

The Big Idea

Loop peeling (step 3)

We apply a classic optimization.

```
1   # Preamble
2   L0(i0):
3   i1 = i0 + 1
4   print(i1)
5   jump(L1, i0)
6
7   # Peeled loop
8   L1(i0):
9   i1 = i0 + 1
10  print(i1)
11  jump(L1, i0)
```

Question: Which optimization?

The Big Idea

Loop peeling (step 3)

Common sub-expression elimination!

```
1   # Preamble
2   L_0(i_0):
3   i_1 = i_0 + 1
4   print(i_1)
5   jump(L_1, i_0)
6
7   # Peeled loop
8   L_1(i_0):
9   ~~i_1 = i_0 + 1~~
10  print(i_1)
11  jump(L_1, i_0)
```

Problem: trace is malformed; i_1 is not local to the peeled loop and not passed as a parameter.

The Big Idea

Loop peeling (step 4)

Pass i_1 from preamble to peeled loop.

```
1   # Preamble
2   L0(i0):
3   i1 = i0 + 1
4   print(i1)
5   jump(L1, i0, i1)
6
7   # Peeled loop
8   L1(i0, i1):
9
10  print(i1)
11  jump(L1, i0, i1)
```

That's it! That's final result, and the whole technique!

The Big Idea

Question: Was that difficult?

The Big Idea

Answer: No[2]

[2]Unless I did a really bad job.

The Big Idea

Question: What were the four steps?

The Big Idea

Answer:

1.

The Big Idea

Answer:

1. Duplicate loop trace
2.

The Big Idea

Answer:

1. Duplicate loop trace
2. Update label names (preamble $\rightarrow$ peeled loop)
3.

The Big Idea

Answer:

1. Duplicate loop trace
2. Update label names (preamble $\rightarrow$ peeled loop)
3. Apply optimization(s)
4.

The Big Idea

Answer:

1. Duplicate loop trace
2. Update label names (preamble $\rightarrow$ peeled loop)
3. Apply optimization(s)
4. Fix jump parameters

Running example

Running example

```
1   class BoxedInteger(Base):
2       def __init__(self, intval):
3           self.intval = intval
4       def add(self, other):
5           return other.add_int(self.intval)
6       def add_int(self, n):
7           return BoxedInteger(n + self.intval)
8       def add_float(self, f):
9           return BoxedFloat(f + float(self.intval))
10
11  class BoxedFloat(Base):
12      def __init__(self, floval):
13          self.floval = floval
14      def add(self, other):
15          return other.add_float(self.floval)
16      def add_int(self, n):
17          return BoxedFloat(float(n) + self.floval)
18      def add_float(self, f):
19          return BoxedFloat(f + self.floval)
20
21  def f(y):
22      step = BoxedInteger(-1)
23      while True:
24          y = y.add(step)
```

Unoptimized trace

```
1   L0(p0, p1):
2   # Inside f: y = y.add(step)
3   guard_class(p1, BoxedInteger)
4       # Inside BoxedInteger.add
5       i2 = get(p1, intval)
6       guard_class(p0, BoxedInteger)
7           # Inside BoxedInteger.add_int
8           i3 = get(p0, intval)
9           i4 = i2 + i3
10          p5 = new(BoxedInteger)
11              # Inside BoxedInteger.__init__
12              set(p5, intval, i4)
13  jump(L0, p0, p5)
```

- $p_0 = \text{step}$
- $p_1 = \text{y}$

Details

Details

Overview

- Input parameters
- List of operations
- Jump operation + jump parameters

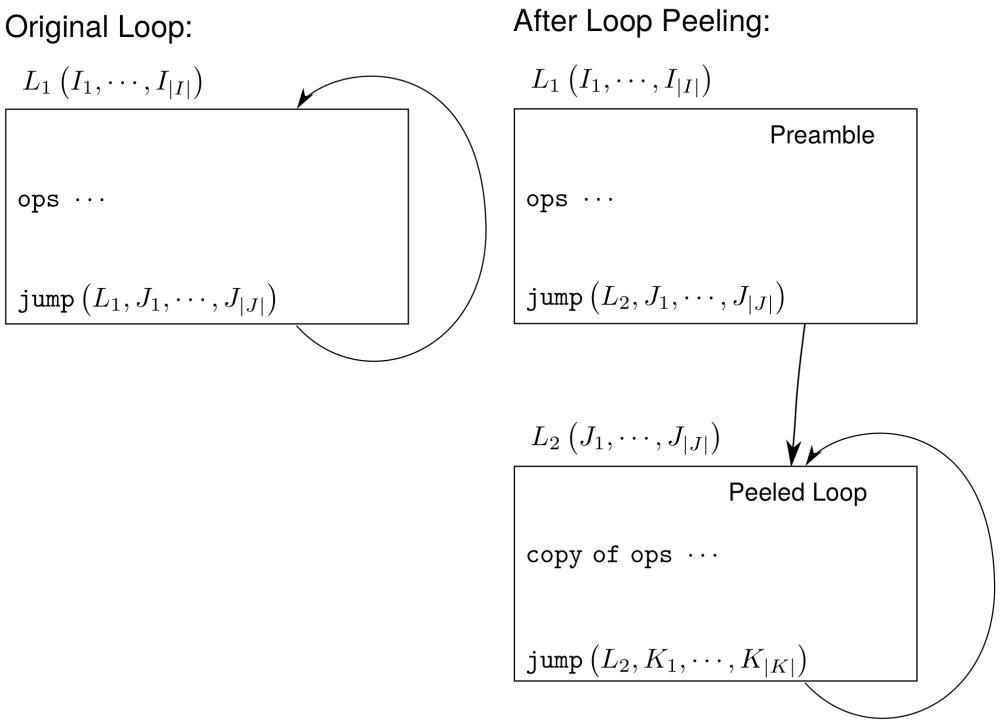

Figure 3. Overview of Loop Peeling

Details

Some definitions

$I = (I_1, I_2, ..., I_m)$: Input parameters of the preamble

$J = (J_1, J_2, ..., J_n)$: $\begin{cases} \text{Jump parameters of the preamble} \\ \text{Input parameters of the peeled loop} \end{cases}$

$K = (K_1, K_2, ..., K_n)$: Jump parameters of peeled loop

m : Preamble Var $\rightarrow$ Peeled Loop Var

Details

Mapping function

```
1        L_0(p_0, p_1):
2        # Inside f: y = y.add(step)
3        guard_class(p_1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i_2 = get(p_1, intval)
6            guard_class(p_0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i_3 = get(p_0, intval)
9                i_4 = i_2 + i_3
10               p_5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p_5, intval, i_4)
13       jump(L_1, p_0, p_5)
14
15       L_1(p_0, p_5):
16       # Inside f: y = y.add(step)
17       guard_class(p_5, BoxedInteger)
18           # Inside BoxedInteger.add
19           i_6 = get(p_5, intval)
20           guard_class(p_0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i_7 = get(p_0, intval)
23               i_8 = i_6 + i_7
24               p_9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p_9, intval, i_8)
27       jump(L_1, p_0, p_9)
```

Details

Mapping function

$$
\begin{aligned}
m(p_0) &= p_0 \\
m(p_1) &= p_5 \\
m(i_2) &= i_6 \\
m(i_3) &= i_7 \\
m(i_4) &= i_8 \\
m(p_5) &= p_9
\end{aligned}
$$

Details

```
1        L_0(p_0, p_1):
2        # Inside f: y = y.add(step)
3        guard_class(p_1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i_2 = get(p_1, intval)
6            guard_class(p_0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i_3 = get(p_0, intval)
9                i_4 = i_2 + i_3
10               p_5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p_5, intval, i_4)
13       jump(L_1, p_0, p_5)
```

- We know that p_5 is a BoxedInteger
- Passed to the peeled loop

Details

```
15      L_1(p_0, p_5):
16      # Inside f: y = y.add(step)
17      guard_class(p_5, BoxedInteger)
18          # Inside BoxedInteger.add
19          i_6 = get(p_5, intval)
20          guard_class(p_0, BoxedInteger)
21              # Inside BoxedInteger.add_int
22              i_7 = get(p_0, intval)
23              i_8 = i_6 + i_7
24              p_9 = new(BoxedInteger)
25                  # Inside BoxedInteger.__init__
26                  set(p_9, intval, i_8)
27      jump(L_1, p_0, p_9)
```

Question: Does the peeled loop need to do a guard_class? Why?

Details

```
15      L_1(p_0, p_5):
16      # Inside f: y = y.add(step)
17      guard_class(p_5, BoxedInteger)
18          # Inside BoxedInteger.add
19          i_6 = get(p_5, intval)
20          guard_class(p_0, BoxedInteger)
21              # Inside BoxedInteger.add_int
22              i_7 = get(p_0, intval)
23              i_8 = i_6 + i_7
24              p_9 = new(BoxedInteger)
25                  # Inside BoxedInteger.__init__
26                  set(p_9, intval, i_8)
27      jump(L_1, p_0, p_9)
```

Answer: No!

Loop-invariant code motion is not the only effect of loop peeling.

Details

Common Sub-expression Elimination

```
1        L0(p0, p1):
2        # Inside f: y = y.add(step)
3        guard_class(p1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i2 = get(p1, intval)
6            guard_class(p0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i3 = get(p0, intval)
9                i4 = i2 + i3
10               p5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p5, intval, i4)
13       jump(L1, p0, p5)
14
15       L1(p0, p5):
16       # Inside f: y = y.add(step)
17
18           # Inside BoxedInteger.add
19           i6 = get(p5, intval)
20           guard_class(p0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i7 = get(p0, intval)
23               i8 = i6 + i7
24               p9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p9, intval, i8)
27       jump(L1, p0, p9)
```

Details

Common Sub-expression Elimination

```
1        L0(p0, p1):
2        # Inside f: y = y.add(step)
3        guard_class(p1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i2 = get(p1, intval)
6            guard_class(p0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i3 = get(p0, intval)
9                i4 = i2 + i3
10               p5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p5, intval, i4)
13       jump(L1, p0, p5, i3)
14
15       L1(p0, p5, i3):
16       # Inside f: y = y.add(step)
17
18           # Inside BoxedInteger.add
19           i6 = get(p5, intval)
20           guard_class(p0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i7 = get(p0, intval)
23               i8 = i6 + i3
24               p9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p9, intval, i8)
27       jump(L1, p0, p9, i3)
```

Details

Common Sub-expression Elimination

We need to keep track of which preamble variables are used in the peeled loop, and pass them as extra parameters in the jump instruction.

Let $H = (H_1, H_2, ..., H_{|H|})$ be the preamble variables used in the peeled loop. The new jump parameters will be:

$$\hat{J} = (J_1, J_2, ..., J_n, H_1, H_2, ..., H_{|H|})$$

The jump parameters of the peeled loop also need to be updated:

$$\hat{K} = (K_1, K_2, ..., K_n, m(H_1), m(H_2), ..., m(H_{|H|}))$$

Details

Allocation Removal

PyPy's allocation removal optimization can identify objects allocated inside the loop that never escape it.

- Optimistically remove all *new* operations
  - If it is discovered that the object escapes, *new* operation will be put back in
- Remove *get*, *set*, *guard_class* operations
- Keep track of attributes

Details

Allocation Removal

```
1        L_0(p_0, p_1):
2        # Inside f: y = y.add(step)
3        guard_class(p_1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i_2 = get(p_1, intval)
6            guard_class(p_0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i_3 = get(p_0, intval)
9                i_4 = i_2 + i_3
10               p_5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p_5, intval, i_4)
13       jump(L_1, p_0, p_5)
14
15       L_1(p_0, p_5):
16       # Inside f: y = y.add(step)
17       guard_class(p_5, BoxedInteger)
18           # Inside BoxedInteger.add
19           i_6 = get(p_5, intval)
20           guard_class(p_0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i_7 = get(p_0, intval)
23               i_8 = i_6 + i_7
24               p_9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p_9, intval, i_8)
27       jump(L_1, p_0, p_9)
```

Details

Allocation Removal

```
1        L_0(p_0, p_1):
2        # Inside f: y = y.add(step)
3        guard_class(p_1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i_2 = get(p_1, intval)
6            guard_class(p_0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i_3 = get(p_0, intval)
9                i_4 = i_2 + i_3
10               p_5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p_5, intval, i_4)
13       jump(L_1, p_0, p_5)
14
15       L_1(p_0, p_5):
16       # Inside f: y = y.add(step)
17       guard_class(p_5, BoxedInteger)
18           # Inside BoxedInteger.add
19           i_6 = get(p_5, intval)
20           guard_class(p_0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i_7 = get(p_0, intval)
23               i_8 = i_6 + i_7
24               p_9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p_9, intval, i_8)
27       jump(L_1, p_0, p_9)
```

Details

Allocation Removal

```
1        L0(p0, p1):
2        # Inside f: y = y.add(step)
3        guard_class(p1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i2 = get(p1, intval)
6            guard_class(p0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i3 = get(p0, intval)
9                i4 = i2 + i3
10               p5 = new(BoxedInteger)
11                   # Inside BoxedInteger.__init__
12                   set(p5, intval, i4)
13       jump(L1, p0, i4)
14
15       L1(p0, i4):
16       # Inside f: y = y.add(step)
17       guard_class(p5, BoxedInteger)
18           # Inside BoxedInteger.add
19           i6 = get(p5, intval)
20           guard_class(p0, BoxedInteger)
21               # Inside BoxedInteger.add_int
22               i7 = get(p0, intval)
23               i8 = i4 + i7
24               p9 = new(BoxedInteger)
25                   # Inside BoxedInteger.__init__
26                   set(p9, intval, i8)
27       jump(L1, p0, i8)
```

Details

Allocation Removal

Let $P^{(k)}$ be the vector containing the attributes of J_k.

$$\tilde{J}^{(k)} = \begin{cases} (J_k) & \text{if } J_k \text{ is concrete} \\ P^{(k)} & \text{if } J_k \text{ is allocation-removed} \end{cases}$$

$$\hat{J} = (\tilde{J}^{(1)}, \tilde{J}^{(2)}, ..., \tilde{J}^{(|J|)})$$
$$\hat{K} = map(m, \hat{J})$$

Details

Allocation Removal

Question: Is this removal of unnecessary allocations a typed-based or untyped unboxing technique?

Details

Allocation Removal

Answer: Untyped.

Details

End result

```
1        L_0(p_0, p_1):
2        # Inside f: y = y.add(step)
3        guard_class(p_1, BoxedInteger)
4            # Inside BoxedInteger.add
5            i_2 = get(p_1, intval)
6            guard_class(p_0, BoxedInteger)
7                # Inside BoxedInteger.add_int
8                i_3 = get(p_0, intval)
9                i_4 = i_2 + i_3
10       jump(L_1, p_0, i_3, i_4)
11
12       L_1(p_0, i_3, i_4):
13               i_8 = i_4 + i_3
14       jump(L_1, p_0, i_3, i_8)
```

Benchmarks

Benchmarks

Configuration

Machine:

- Intel Xeon X5680 @3.33 GHz
- 12M cache
- 16 GB of RAM
- Ubuntu Linux 11.04, 64-bit

Software:

- PyPy 1.9
- CPython 2.7.1
- GCC 4.5.2 (shipped with Ubuntu 11.04)
- LuaJIT 2.0 beta (git rev: 0dd175d9)

Benchmarks

Tests

SciMark:

- FFT(n, c): Fast Fourier Transform, n elements, repeated c times
- LU(n, c): LU factorization of a $n \times n$ matrix, repeated c times
- MonteCarlo(n): Monte Carlo integration with n random points
- SOR(n, c): Jacobi successive over-relaxation on a $n \times n$ grid, repeated c times
- SparseMatMul(n, z, c): $n \times n$ sparse matrix multiplied by vector, containing z non-zero elements, repeated c times

Benchmarks

Tests

Numerical calculations:

- conv3(n): one-dimensional convolution, fixed kernel-sized 3
- conv3x3(n, m): two-dimensional convolution, fixed kernel-sized 3×3
- dilate3x3(n): two-dimensional dilation, kernel fixed-sized 3×3
- sobel(n): video algorithm, finds edges in an image
- sqrt(n): approximate $\sqrt{n}$

Benchmarks

Results

Ratios vs C implementation

Benchmark	CPython	PyPy	PyPy LP	LuaJIT	LuaJIT LP	GCC -O3
FFT(1024,32768)	335.05	14.88	9.09	3.18	1.96	1.00
FFT(1048576,2)	71.00	4.96	2.47	1.51	1.29	1.00
LU(100,4096)	1,484.32	24.23	10.07	6.44	1.14	1.00
LU(1000,2)	1,469.71	23.05	9.22	6.15	1.03	1.00
MonteCarlo(268435456)	366.21	12.19	9.07	2.32	1.67	1.00
SOR(100,32768)	828.48	4.68	1.51	1.15	0.74	1.00
SOR(1000,256)	812.38	4.35	1.41	1.09	0.72	1.00
SparseMatMult(...)	201.99	13.18	8.98	5.27	2.44	1.00
SparseMatMult(...)	197.44	14.18	7.29	5.99	2.03	1.00
conv3(1e6)	82.00	1.88	0.85	1.17	0.30	1.00
conv3x3(1000,1000)	817.35	4.12	1.18	1.29	0.88	1.00
dilate3x3(1000,1000)	808.94	25.59	23.00	1.29	0.94	1.00
sobel(1000,1000)	611.88	2.88	1.24	2.18	1.41	1.00
sqrt(float)	17.64	1.61	1.05	1.25	0.98	1.00
sqrt(int)	11.13	2.58	2.12			1.00
sqrt(Fix16)	345.87	3.82	2.21	2.99	1.10	1.00
Average	528.84	9.89	5.67	2.88	1.24	1.00

</presentation>