Loop-Aware Optimizations in PyPy's Tracing JIT

Håkan Ardö
Centre for Mathematical Sciences, Lund University
hakan@debian.org

Carl Friedrich Bolz
Heinrich-Heine-Universität Düsseldorf
cfbolz@gmx.de

Maciej Fijałkowski
fijall@gmail.com

Abstract

One of the nice properties of a tracing just-in-time compiler (JIT) is that many of its optimizations are simple, requiring one forward pass only. This is not true for loop-invariant code motion which is a very important optimization for code with tight kernels. Especially for dynamic languages that typically perform quite a lot of loop invariant type checking, boxed value unwrapping and virtual method lookups.

In this paper we explain a scheme pioneered within the context of the LuaJIT project for making basic optimizations loop-aware by using a simple pre-processing step on the trace without changing the optimizations themselves.

We have implemented the scheme in RPython's tracing JIT compiler. PyPy's Python JIT executing simple numerical kernels can become up to two times faster, bringing the performance into the ballpark of static language compilers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, incremental compilers, interpreters, run-time environments

General Terms Languages, Performance, Experimentation

Keywords Tracing JIT, Optimization, Loop-Invariant Code Motion

1. Introduction

A dynamic language typically needs to do quite a lot of type checking, wrapping/unwrapping of boxed values, and virtual method dispatching. For tight computationally intensive loops a significant amount of the execution time might be spent on such tasks instead of the actual computations. Moreover, the type checking, unwrapping and method lookups are often loop invariant and performance could be increased by moving those operations out of the loop. We explain a simple scheme to make a tracing JIT loop-aware by allowing it's existing optimizations to perform loop invariant code motion.

One of the advantages that tracing just-in-time compilers (JITs) have above traditional method-based JITs is that their optimizers are much easier to write. Because a tracing JIT produces only linear pieces of code without control flow joins, many optimization passes on traces can have a very simple structure: They often consist of one forward pass replacing operations by faster ones or even discarding them as they walk along it. This makes optimization of traces very similar to symbolic execution. Also, many difficult problems in traditional optimizers become tractable if the optimizer does not need to deal with control flow merges.

One disadvantage of this simplicity is that such forward-passing optimizers ignore the only bit of control flow they have available, which is the fact that most traces actually represent loops. Making use of this information is necessary to perform optimizations that take the whole loop into account, such as loop-invariant code motion or optimizations that improve several iterations of the loop. Having to deal with this property of traces complicates the optimization passes, as a more global view of a trace needs to be considered when optimizing.

Mike Pall pioneered a solution to address this problem in the context of a dynamic language using a tracing JIT compiler. He published his algorithm and its rationale in 2009 [19] and implemented it in LuaJIT 2.0[1], an open source JIT compiler for the Lua language. His approach allows to reuse all forward pass optimizations to achieve loop invariant code motion and other loop-related optimizations, which greatly simplifies the implementation. We have implemented the same approach in RPython's tracing JIT compiler, the results of which we present here.

The resulting optimizations one gets using this scheme are in no way novel, most of them are well-known loop optimizations. However, the way to implement them is a lot simpler than directly implementing loop-aware optimizations.

2. Background: RPython and PyPy

The work described in this paper was done in the context of the PyPy project.[2] PyPy is a framework for implementing dynamic languages efficiently [20]. When implementing a language with PyPy, one writes an interpreter for the language in RPython [2]. RPython ("Restricted Python") is a subset of Python chosen in such a way that it can be efficiently translated to a C-based virtual machine (VM) by performing type inference.

Many low-level aspects of the final VM are not contained within the interpreter implementation but are inserted during translation to C. Examples for this are a garbage collector and also a tracing JIT compiler [6].

RPython's tracing JIT compiler traces on the level of RPython programs. Thus it actually traces the execution of an interpreter written in RPython, not of the program itself. This makes the details of the object model of the implemented language transparent and optimizable by the tracing JIT. In the context of this paper, this aspect of RPython's tracing JIT can be ignored. Instead, it is sufficient to view RPython's tracing JIT as a JIT for RPython.

DLS'12, October 22, 2012, Tucson, Arizona, USA.

[1] http://luajit.org/

[2] http://pypy.org

3. Motivation

To motivate the approach we propose here, let's look at a trivial (unrealistic) trace which corresponds to an infinite loop:

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

The first line is a label L_0 with argument i_0. Every label has a list of arguments. The `print` operation just prints its argument (it is not an operation that RPython's tracing JIT really supports, we just use it for this example). The `jump` operation jumps back to the beginning of the trace, listing the new values of the arguments of the trace. In this case, the new value of i_0 is i_0, making it a loop-invariant.

Because i_0 is loop-invariant, the addition could be moved out of the loop. However, it is desirable to get this effect using our existing optimization passes without changing them too much. Optimizations with one forward pass cannot directly achieve this effect: They just look at the trace without taking into account that the trace executes many times in a row. Therefore to achieve loop-invariant code motion, we peel one iteration off the loop before running the optimizations. This peeling gives the following trace:

```
L_0(i_0):                          1
i_1 = i_0 + 1                      2
print(i_1)                         3
jump(L_1, i_0)                     4
                                   5
L_1(i_0):                          6
i_2 = i_0 + 1                      7
print(i_2)                         8
jump(L_1, i_0)                     9
```

The iteration of the loop that was peeled off (lines 1-4) is called the *preamble*, the loop afterwards (lines 6-9) the *peeled loop*.

Now the optimizer optimizes both of these two iterations of the loop together, disregarding the `jump` and the label in lines 4-6. Doing this, common subexpression elimination will discover that the two additions are the same, and replace i_2 with i_1. This leads to the following trace:

```
L_0(i_0):                          1
i_1 = i_0 + 1                      2
print(i_1)                         3
jump(L_1, i_0)                     4
                                   5
L_1(i_0):                          6
print(i_1)                         7
jump(L_1, i_0)                     8
```

This trace is malformed, because i_1 is used after the label L_1 without being passed there, so we need to add i_1 as an argument to the label and pass it along the `jump`s:

```
L_0(i_0):                          1
i_1 = i_0 + 1                      2
print(i_1)                         3
jump(L_1, i_0, i_1)                4
                                   5
L_1(i_0, i_1):                     6
print(i_1)                         7
jump(L_1, i_0, i_1)                8
```

The final result is that the loop-invariant code was moved out of the loop into the peeled-off iteration. Thus the addition is only executed in the first iteration, while the result is reused in all further iterations.

This scheme is quite powerful and generalizes to other optimizations than just common subexpression elimination. It allows linear optimization passes to perform loop-aware optimizations, such as loop-invariant code motion without changing them at all. All that is needed is to peel off one iteration, then apply one-pass optimizations and make sure that the necessary extra arguments are inserted into the label of the loop itself and the jumps afterwards.

This is the key insight of the implementation scheme: If an optimization is given two iterations together at the same time, the optimization has enough context to remove operations from the peeled loop, because it detects that the operation was performed in the preamble already. Thus at runtime these moved operations are only executed once when entering the loop and the results are reused in further iterations.

```
class Base(object):
   pass

class BoxedInteger(Base):
   def __init__(self, intval):
      self.intval = intval

   def add(self, other):
      return other.add__int(self.intval)

   def add__int(self, intother):
      return BoxedInteger(intother + self.intval)

   def add__float(self, floatother):
      floatvalue = floatother + float(self.intval)
      return BoxedFloat(floatvalue)


class BoxedFloat(Base):
   def __init__(self, floatval):
      self.floatval = floatval

   def add(self, other):
      return other.add__float(self.floatval)

   def add__int(self, intother):
      floatvalue = float(intother) + self.floatval
      return BoxedFloat(floatvalue)

   def add__float(self, floatother):
      return BoxedFloat(floatother + self.floatval)


def f(y):
   step = BoxedInteger(-1)
   while True:
      y = y.add(step)
```

Figure 1. An "Interpreter" for a Tiny Dynamic Language Written in RPython

4. Running Example

The last section gave a motivating but unrealistically small example. This section will define a slightly larger example that the rest of the paper uses to demonstrate the effect of optimizations. For this we are going to use a tiny interpreter for a dynamic language with a very small object model, that just supports an integer and a float type (this example has been taken from a previous paper [4]). The objects support only one operation, `add`, which adds two objects (promoting ints to floats in a mixed addition). The implementation of `add` uses classical double-dispatching. The classes can be seen in Figure 1 (written in RPython).

Using these classes to implement arithmetic shows the basic problem of many dynamic language implementations. All the numbers are instances of either `BoxedInteger` or `BoxedFloat`, therefore they consume space on the heap. Performing many arithmetic operations produces lots of garbage quickly, putting pressure on the garbage collector. Using double dispatching to implement the nu-

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

Figure 2. An Unoptimized Trace of the example interpreter

meric tower needs two method calls per arithmetic operation, which is costly due to the method dispatch.

Let us now consider an "interpreter" function `f` that uses the object model (see the bottom of Figure 1). Simply running this function is slow, because there are lots of virtual method calls inside the loop, two for each call to `add`. These method calls need to check the type of the involved objects every iteration. In addition, a lot of objects are created when executing that loop, many of these objects are short-lived. The actual computation that is performed by `f` is simply a sequence of float or integer additions (note that `f` does not actually terminate, but it is still instructive to look at the produced traces).

If the function is executed using the tracing JIT, with `y` being a `BoxedInteger`, the produced trace looks like the one of Figure 2 (lines starting with a hash "#" are comments). The trace corresponds to one iteration of the while-loop in `f`.

The operations in the trace are indented corresponding to the stack level of the function that contains the traced operation. The trace is in single-assignment form, meaning that each variable is assigned a value exactly once. The arguments p_0 and p_1 of the loop correspond to the live variables `y` and `step` in the while-loop of the original function.

The label of the loop is L_0 and is used by the jump instruction to identify it's jump target.

The operations in the trace correspond to the operations in the RPython program in Figure 1:

- `new` creates a new object.
- `get` reads an attribute of an object.
- `set` writes to an attribute of an object.
- `guard_class` is a precise type check, not checking for subclasses.

Inlined method calls in the trace are preceded by a `guard_class` operation, to check that the class of the receiver is the same as the one that was observed during tracing. These guards make the trace specific to the situation where `y` is really a `BoxedInteger`. When the trace is turned into machine code and afterwards executed with `BoxedFloat`, the first `guard_class` instruction will fail and execution will continue using the interpreter.

5. Making Trace Optimizations Loop Aware

Before a trace is compiled to machine code by the backend, it is optimized to achieve better performance. One goal of that is to move operations out of the loop to execute them only once and not every iteration. This can be achieved by loop peeling. It leaves the loop body intact, but prefixes it with one iteration of the loop. This operation by itself will not achieve anything. But if it is combined with other optimizations it can increase the effectiveness of those optimizations. For many optimizations of interest only a few additional details have to be considered when they are combined with loop peeling. These are described below by explaining the loop peeling optimization followed by a set of other optimizations and how they interact with loop peeling.

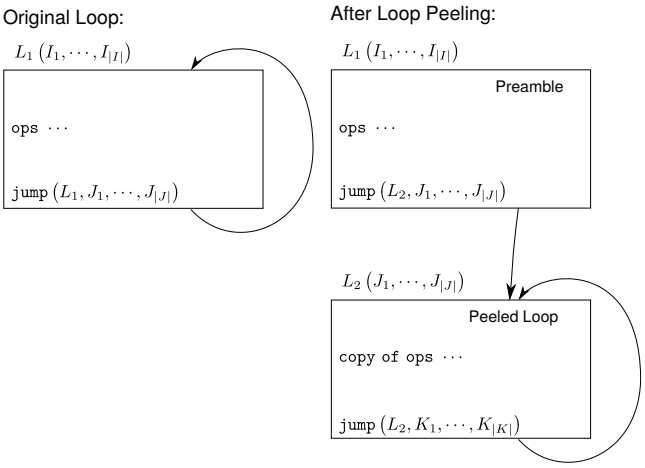

Figure 3. Overview of Loop Peeling

5.1 Loop Peeling

Loop peeling is achieved by appending a copy of the traced iteration at the end of itself. See Figure 3 for an illustration. The first part (called *preamble*) finishes with a jump to the second part (called the *peeled loop*). The second part finishes with a jump to itself. This way the preamble will be executed only once while the peeled loop will be used for every further iteration. New variable names have to be introduced in the entire copied trace in order to maintain the SSA-property.

When peeling the loop, no assumptions are made that the preamble is the *first* iteration, when later executing the loop. The preamble stays general enough to correspond to any iteration of the loop. However, the peeled loop can then be optimized using the assumption that a previous iteration (the preamble) has been executed already.

When applying optimizations to this two-iteration trace some care has to taken as to how the arguments of the two `jump` operations and the input arguments of the peeled loop are treated. It has to be ensured that the peeled loop stays a proper trace in the sense that the operations within it only operate on variables that are either among its input arguments or produced within the peeled loop. To ensure this we need to introduce a bit of formalism.

The original trace (prior to peeling) consists of three parts. A vector of input variables, $I = \left(I_1, I_2, \cdots, I_{|I|}\right)$, a list of non- jump operations and a single jump operation. The jump operation contains a vector of jump variables, $J = \left(J_1, J_2, \cdots, J_{|J|}\right)$, that are passed as the input variables of the target loop. After loop peeling there will be a second copy of this trace with input variables equal to the jump arguments of the preamble, J, and jump arguments K. Figure 3 illustrates the general case. The running example in Figure 2 has $I = (p_0, p_1)$ and $J = (p_0, p_5)$. The result of applying loop peeling to it is shown in Figure 4 with $K = (p_0, p_9)$.

To construct the second copy of the trace (the peeled loop) from the first (the preeamble) we need a function m, mapping the variables of the preamble onto the variables of the peeled loop. This function is constructed during the copying. It is initialized by mapping the input arguments, I, to the jump arguments J,

$$m\left(I_i\right) = J_i \text{ for } i = 1, 2, \cdots |I|. \quad (1)$$

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

Figure 4. A peeled trace of the example interpreter

In the example that means:

$$\begin{array}{rcl} m\left(p_0\right) & = & p_0 \\ m\left(p_1\right) & = & p_5 \end{array} . \tag{2}$$

Each operation in the trace is copied in order. To copy an operation $v = \text{op}\left(A_1, A_2, \cdots, A_{|A|}\right)$ a new variable, $\hat{v}$, is introduced. The copied operation will return $\hat{v}$ using

$$\hat{v} = \text{op}\left(m\left(A_1\right), m\left(A_2\right), \cdots, m\left(A_{|A|}\right)\right). \tag{3}$$

Before the next operation is copied, m is extend by assigning $m\left(v\right) = \hat{v}$. For the example above, that will extend m with

$$\begin{array}{rcl} m\left(i_2\right) & = & i_6 \\ m\left(i_3\right) & = & i_7 \\ m\left(i_4\right) & = & i_8 \\ m\left(p_5\right) & = & p_9 \end{array} . \tag{4}$$

6. Interaction of Optimizations with Loop Peeling

6.1 Redundant Guard Removal

Redundant guard removal removes guards that are implied by other guards earlier in the trace. The most common case is the removal of a guard that has already appeared. No special concern needs to be taken when implementing redundant guard removal together with loop peeling. The guards from the preamble might make the guards of the peeled loop redundant and thus removed. Therefore one effect of combining redundant guard removal with loop peeling is that loop-invariant guards are moved out of the loop. The peeled loop of the example reduces to the trace in Figure 5.

The guard on p_5 on line 17 of Figure 4 can be removed since p_5 is allocated on line 10 with a known class. The guard on p_0 on line 20 can be removed since it is identical to the guard on line 6.

Note that the guard on p_5 is removed even though p_5 is not loop invariant, which shows that loop invariant code motion is not the only effect of loop peeling. Loop peeling can also remove guards that are implied by the guards of the previous iteration.

```
L_1(p_0, p_5):                                          1
# inside f: y = y.add(step)                             2
    # inside BoxedInteger.add                           3
    i_6 = get(p_5, intval)                              4
        # inside BoxedInteger.add__int                  5
        i_7 = get(p_0, intval)                          6
        i_8 = i_6 + i_7                                 7
        p_9 = new(BoxedInteger)                         8
            # inside BoxedInteger.__init__              9
            set(p_9, intval, i_8)                       10
jump(L_1, p_0, p_9)                                     11
```

Figure 5. Peeled loop after redundant guard removal

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

Figure 6. Trace after common subexpression elimination

6.2 Common Subexpression Elimination and Heap Optimizations

If a pure operation appears more than once in the trace with the same input arguments, it only needs to be executed the first time and then the result can be reused for all other appearances. This is achieved by common subexpression elimination. RPython's optimizers can also remove repeated heap reads if the intermediate operations cannot have changed their value.[3]

When that is combined with loop peeling, the single execution of the operation is placed in the preamble. That is, loop invariant pure operations and heap reads are moved out of the loop.

Consider the `get` operation on line 22 of Figure 4. The result of this operation can be deduced to be i_3 from the `get` operation on line 8. The optimization will thus remove line 22 from the trace and replace i_7 with i_3. Afterwards the trace is no longer in the correct form, because the argument i_3 is not passed along the loop arguments. Therefore i_3 needs to be added to the loop arguments.

Doing this, the trace from Figure 4 will be optimized to the trace in Figure 6.

After loop peeling and redundant operation removal the peeled loop will typically no longer be in SSA form but operate on variables

[3] We perform a type-based alias analysis to know which writes can affect which reads [11]. In addition writes on newly allocated objects can never change the value of old existing ones.

that are the result of operations in the preamble. The solution is to extend the input arguments, J, with those variables. This will also extend the jump arguments of the preamble, which is also J. Implicitly that also extends the jump arguments of the peeled loop, K, since they are the image of J under m. For the example I has to be replaced by $\hat{I}$ which is formed by appending i_3 to I. At the same time K has to be replaced by $\hat{K}$ which is formed by appending $m(i_3) = i_7$ to K. The variable i_7 will then be replaced by i_3 by the heap caching optimization as it has removed the variable i_7.

In general what is needed is to keep track of which variables from the preamble are reused in the peeled loop. By constructing a vector, H, of such variables, the input and jump arguments can be updated using

$$\hat{J} = \left(J_1, J_2, \cdots, J_{|J|}, H_1, H_2, \cdots, H_{|H|}\right) \tag{5}$$

and

$$\hat{K} = \left(K_1, K_2, \cdots, K_{|J|}, m(H_1), m(H_2), \cdots, m(H_{|H|})\right). \tag{6}$$

In the optimized trace J is replaced by $\hat{J}$ and K by $\hat{K}$.

It is interesting to note that the described approach deals correctly with implicit control dependencies, whereas in other approaches this needs to be carefully programmed in. A commonly used example for a control dependency is a division operation that needs to be preceded by a check for the second argument being 0. In a trace, such a check would be done with a guard. The division operation must not be moved before that guard, and indeed, this is never done. If the division is loop invariant, the result computed by the copy of the division operation in the preamble is reused. This division operation is preceded by a copy of the guard that checks that the second argument is not 0, which ensures that the division can be executed correctly. Such control dependencies are common in traces produced by dynamic languages. Reading a field out of an object is often preceded by checking the type of the object.

6.3 Allocation Removal

RPython's allocation removal optimization [4] makes it possible to identify objects that are allocated within the loop but never escape it. That is, no outside object ever gets a reference to them. This is performed by processing the operations in order and optimistically removing every `new` operation. Later on if it is discovered that a reference to the object escapes the loop, the `new` operation is inserted at this point. All operations (`get`, `set` and `guard_class`) on the removed objects are also removed and the optimizer needs to keep track of the value of all used attributes of the object.

Consider again the original unoptimized trace of Figure 4. Line 10 contains the first allocation. It is removed and p_5 is marked as allocation-removed. This means that it refers to an object that has not yet been (and might never be) allocated. Line 12 sets the `intval` attribute of p_5. This operation is also removed and the optimizer registers that the attribute `intval` of p_5 is i_4.

When the optimizer reaches line 13 it needs to construct the arguments of the `jump` operation, which contains the reference to the allocation-removed object in p_5. This can be achieved by exploding p_5 into the attributes of the allocation-removed object. In this case there is only one such attribute and its value is i_4, which means that p_5 is replaced with i_4 in the jump arguments.

In the general case, each allocation-removed object in the jump arguments is exploded into a vector of variables containing the values of all registered attributes.[4] If some of the attributes are themselves references to allocation-removed objects they are recursively exploded to make the vector contain only concrete variables. Some care has to be taken to always place the attributes in the same order when performing this explosion. Notation becomes somewhat simpler if every concrete variable of the jump arguments is also exploded into a vector containing itself. For every variable, J_k, of the original jump arguments, J, let

$$\tilde{J}^{(k)} = \begin{cases} (J_k) & \text{if } J_k \text{ is concrete} \\ H^{(k)} & \text{if } J_k \text{ is allocation-removed} \end{cases}, \tag{7}$$

where $H^{(k)}$ is a vector containing all concrete attributes of J_k. The arguments of the optimized `jump` operation are constructed as the concatenation all the $\tilde{J}^{(k)}$ vectors,

$$\hat{J} = \left(\tilde{J}^{(1)} \quad \tilde{J}^{(2)} \quad \cdots \quad \tilde{J}^{(|J|)} \right). \tag{8}$$

The arguments of the `jump` operation of the peeled loop, K, is constructed from $\hat{J}$ using the map m,

$$\hat{K} = \left(m\left(\hat{J}_1\right), m\left(\hat{J}_1\right), \cdots, m\left(\hat{J}_{|\hat{J}|}\right)\right). \tag{9}$$

In the optimized trace J is replaced by $\hat{J}$ and K by $\hat{K}$. The trace from Figure 2 will be optimized to the trace in Figure 7.

If all the optimizations presented above are applied, the resulting loop looks as in Figure 8. The resulting optimized peeled loop consists of a single integer addition. That is it will become type-specialized to the types of the variables `step` and `y`, and the overhead of using boxed values is removed.

```
L0(p0, p1):                                          1
# inside f: y = y.add(step)                          2
guard_class(p1, BoxedInteger)                        3
    # inside BoxedInteger.add                        4
    i2 = get(p1, intval)                             5
    guard_class(p0, BoxedInteger)                    6
        # inside BoxedInteger.add__int               7
        i3 = get(p0, intval)                         8
        i4 = i2 + i3                                 9
            # inside BoxedInteger.__init__           10
jump(L1, p0, i4)                                     11
                                                     12
L1(p0, i4):                                          13
# inside f: y = y.add(step)                          14
    # inside BoxedInteger.add                        15
    guard_class(p0, BoxedInteger)                    16
        # inside BoxedInteger.add__int               17
        i7 = get(p0, intval)                         18
        i8 = i4 + i7                                 19
            # inside BoxedInteger.__init__           20
jump(L1, p0, i8)                                     21
```

Figure 7. Trace after allocation removal

```
L0(p0, p1):                                          1
# inside f: y = y.add(step)                          2
guard_class(p1, BoxedInteger)                        3
    # inside BoxedInteger.add                        4
    i2 = get(p1, intval)                             5
    guard_class(p0, BoxedInteger)                    6
        # inside BoxedInteger.add__int               7
        i3 = get(p0, intval)                         8
        i4 = i2 + i3                                 9
            # inside BoxedInteger.__init__           10
jump(L1, p0, i4)                                     11
                                                     12
L1(p0, i3, i4):                                      13
    i8 = i4 + i3                                     14
jump(L1, p0, i3, i8)                                 15
```

Figure 8. The fully optimized loop of the example interpreter

[4] This is sometimes called *scalar replacement* [16].

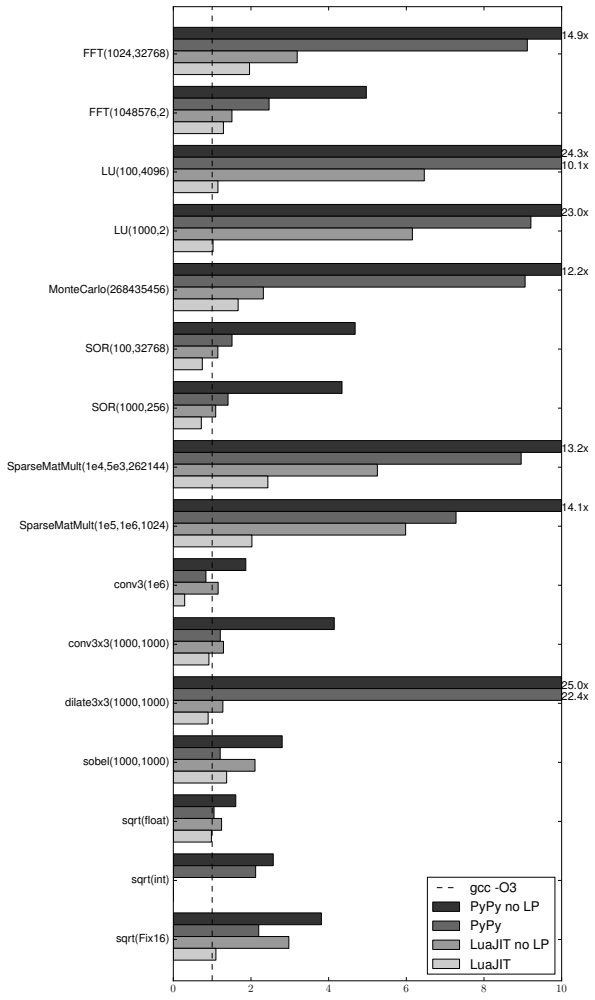

Figure 10. Benchmark results normalized to the runtime of the C version. The CPython results have been omitted to make the plot readable.

7. Benchmarks

The loop peeling optimization was implemented in RPython's tracing JIT in about 450 lines of RPython code. That means that the JIT-compilers generated for all interpreters implemented with RPython now can take advantage of it. Benchmarks have been executed for a few different interpreters and we see improvements in several cases.

An example of an RPython interpreter that is helped greatly by this optimization is our Prolog interpreter [7]. Prolog programs often contain tight loops that perform for example list processing. Furthermore we experimented with a Python library for writing numerical kernels doing array manipulation.

The ideal loop for this optimization is short and contains numerical calculations with no failing guards and no external calls. Larger loops involving many operations on complex objects typically benefit less from it. Loop peeling never makes the generated code worse, in the worst case the peeled loop is exactly the same as the preamble. Therefore we chose to present benchmarks of small numeric kernels where loop peeling can show its use.

The Python interpreter of the RPython framework is a complete Python version 2.7 compatible interpreter. A set of numerical calculations were implemented in both Python, C and Lua and their runtimes are compared in Figure 10 and Figure 9.[5] For benchmarks using larger Python applications the times are unaffected or only slightly improved by the loop optimization of this paper.

The benchmarks are

- **conv3**(n): one-dimensional convolution with fixed kernel-size 3. A single loop is used to calculate a vector $\mathbf{b} = (b_1, \cdots, b_{n-2})$ from a vector $\mathbf{a} = (a_1, \cdots, a_n)$ and a kernel $\mathbf{k} = (k_1, k_2, k_3)$ using $b_i = k_3 a_i + k_2 a_{i+1} + k_1 a_{i+2}$ for $1 \leq i \leq n-2$. Both the output vector, $\mathbf{b}$, and the input vectors, $\mathbf{a}$ and $\mathbf{k}$, are allocated prior to running the benchmark. It is executed with $n = 10^5$.
- **conv3x3**(n, m): two-dimensional convolution with a kernel of fixed size 3×3 using a custom class to represent two-dimensional arrays. It is implemented as two nested loops that iterates over the elements of the $m \times n$ output matrix $\mathbf{B} = (b_{i,j})$ and calculates each element from the input matrix $\mathbf{A} = (a_{i,j})$ and a kernel $\mathbf{K} = (k_{i,j})$ using $b_{i,j} =$

$$\begin{array}{lllllll} k_{3,3}a_{i-1,j-1} & + & k_{3,2}a_{i-1,j} & + & k_{3,1}a_{i-1,j+1} & + \\ k_{2,3}a_{i,j-1} & + & k_{2,2}a_{i,j} & + & k_{2,1}a_{i,j+1} & + \\ k_{1,3}a_{i+1,j-1} & + & k_{1,2}a_{i+1,j} & + & k_{1,1}a_{i+1,j+1} & \end{array} \tag{10}$$

  for $2 \leq i \leq m-1$ and $2 \leq j \leq n-1$. The memory for storing the matrices are again allocated outside the benchmark and $(n, m) = (1000, 1000)$ was used.
- **dilate3x3**(n): two-dimensional dilation with a kernel of fixed size 3×3. This is similar to convolution but instead of summing over the terms in Equation 10, the maximum over those terms is taken. That places a external call to a max function within the loop that prevents some of the optimizations for PyPy.
- **sobel**(n): a low-level video processing algorithm used to locate edges in an image. It calculates the gradient magnitude using sobel derivatives. A Sobel x-derivative, D_x, of a $n \times n$ image, I, is formed by convolving I with

$$K = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \tag{11}$$

  and a Sobel y-derivative, D_y, is formed convolving I with $K^\top$. The gradient magnitude is then formed for each pixel independently by $\sqrt{D_x^2 + D_y^2}$. The two convolutions and the pixelwise magnitude calculation are combined in the implementation of this benchmark and calculated in a single pass over the input image. This single pass consists of two nested loops with a somewhat larger amount of calculations performed each iteration as compared to the other benchmarks.
- **sqrt**(T): approximates the square root of y. The approximation is initialized to $x_0 = y/2$ and the benchmark consists of a single loop updating this approximation using $x_i = (x_{i-1} + y/x_{i-1})/2$ for $1 \leq i < 10^8$. Only the latest calculated value x_i is kept alive as a local variable within the loop. There are three different versions of this benchmark where x_i is represented with different type T of objects: int's, float's and Fix16's. The latter, Fix16, is a custom class that implements fixpoint arithmetic with 16 bits precision. In Python and Lua there is only a single implementation of the benchmark that gets specialized depending on the class of it's input argument, y. In C, there are three different implementations.

[5] The benchmarks and the scripts to run them can be found in the repository for this paper: `https://bitbucket.org/pypy/extradoc/src/tip/talk/dls2012/benchmarks`

	CPython	PyPy no LP	PyPy	LuaJIT no LP	LuaJIT	GCC -O3
FFT(1024,32768)	469.07	20.83 ± 0.039	12.73 ± 0.029	4.45 ± 0.019	2.74 ± 0.021	1.40 ± 0.082
FFT(1048576,2)	58.93	4.12 ± 0.020	2.05 ± 0.007	1.25 ± 0.019	1.07 ± 0.050	0.83 ± 0.044
LU(100,4096)	1974.14	32.22 ± 0.281	13.39 ± 0.063	8.57 ± 0.018	1.52 ± 0.010	1.33 ± 0.070
LU(1000,2)	955.31	14.98 ± 0.436	5.99 ± 0.416	4.00 ± 0.018	0.67 ± 0.014	0.65 ± 0.077
MonteCarlo(268435456)	618.89	20.60 ± 0.097	15.33 ± 0.163	3.92 ± 0.013	2.82 ± 0.010	1.69 ± 0.096
SOR(100,32768)	1458.12	8.24 ± 0.002	2.66 ± 0.002	2.02 ± 0.011	1.31 ± 0.010	1.76 ± 0.088
SOR(1000,256)	1210.45	6.48 ± 0.007	2.10 ± 0.005	1.63 ± 0.006	1.08 ± 0.014	1.49 ± 0.042
SparseMatMult(1e4,5e3,262144)	371.66	24.25 ± 0.074	16.52 ± 0.077	9.69 ± 0.033	4.49 ± 0.036	1.84 ± 0.061
SparseMatMult(1e5,1e6,1024)	236.93	17.01 ± 0.025	8.75 ± 0.149	7.19 ± 0.019	2.43 ± 0.031	1.20 ± 0.053
conv3(1e6)	49.20	1.13 ± 0.043	0.51 ± 0.008	0.70 ± 0.009	0.18 ± 0.009	0.60 ± 0.064
conv3x3(1000,1000)	138.95	0.70 ± 0.007	0.20 ± 0.009	0.22 ± 0.009	0.15 ± 0.010	0.17 ± 0.079
dilate3x3(1000,1000)	137.52	4.35 ± 0.014	3.91 ± 0.037	0.22 ± 0.008	0.16 ± 0.010	0.17 ± 0.061
sobel(1000,1000)	104.02	0.49 ± 0.009	0.21 ± 0.004	0.37 ± 0.014	0.24 ± 0.017	0.17 ± 0.061
sqrt(float)	14.99	1.37 ± 0.001	0.89 ± 0.000	1.06 ± 0.010	0.83 ± 0.014	0.85 ± 0.088
sqrt(int)	13.91	3.22 ± 0.033	2.65 ± 0.001	-	-	1.25 ± 0.053
sqrt(Fix16)	463.46	5.12 ± 0.005	2.96 ± 0.007	4.00 ± 0.040	1.47 ± 0.014	1.34 ± 0.061

Figure 9. Benchmark results in seconds with 95% confidence intervals. The leftmost column gives the name of each benchmark and the values of the benchmark parameters used. The different benchmarks and the meaning of their parameters are described in Section 7.

The Fix16 type is a custom class with operator overloading in Lua and Python. The C version uses a C++ class. The goal of this variant of the benchmark is to check how large the overhead of a custom arithmetic class is, compared to builtin data types.

In Lua there is no direct support for integers so the int version is not provided.

The sobel and conv3x3 benchmarks are implemented on top of a custom two-dimensional array class. It is a straightforward implementation providing 2 dimensional indexing with out of bounds checks and data stored in row-major order. For the C implementations it is implemented as a C++ class. The other benchmarks are implemented in plain C. All the benchmarks except sqrt operate on C double-precision floating point numbers, both in the Python, C and Lua code.

In addition we also ported the SciMark[6] benchmarts to Python, and compared their runtimes with the already existing Lua[7] and C implementations.

SciMark consists of:

- **FFT**(n, c): Fast Fourier Transform of a vector with n elements, represented as an array, repeated c times.
- **LU**(n, c): LU factorization of an $n \times n$ matrix. The rows of the matrix is shuffled which makes the previously used two-dimensional array class unsuitable. Instead a list of arrays is used to represent the matrix. The calculation is repeated c times.
- **MonteCarlo**(n): Monte Carlo integration by generating n points uniformly distributed over the unit square and computing the ratio of those within the unit circle.
- **SOR**(n, c): Jacobi successive over-relaxation on a $n \times n$ grid repreated c times. The same custom two-dimensional array class as described above is used to represent the grid.
- **SparseMatMult**(n, z, c): Matrix multiplication between a $n \times n$ sparse matrix, stored in compressed-row format, and a full storage vector, stored in a normal array. The matrix has z non-zero elements and the calculation is repeated c times.

Benchmarks were run on Intel Xeon X5680 @3.33GHz with 12M cache and 16G of RAM using Ubuntu Linux 11.4 in 64bit mode. The machine was otherwise unoccupied. We used the following software for benchmarks:

- PyPy 1.9
- CPython 2.7.1
- GCC 4.5.2 shipped with Ubuntu 11.4
- LuaJIT 2.0 beta, git head of August 15, 2012, commit ID 0dd175d9

We ran GCC with -O3 -march=native, disabling the automatic loop vectorization. In all cases, SSE2 instructions were used for floating point operations. We also ran PyPy and LuaJIT with loop peeling optimization and without (but otherwise identical).

For PyPy and LuaJIT, 10 iterations were run, prefaced with 3 iterations for warming up. Due to benchmarks taking large amounts of time on CPython, only one run was performed. For GCC, 5 iterations were run. In all cases, the standard deviation is very low, making benchmarks very well reproducible.

We can observe that PyPy (even without loop peeling) is orders of magnitude faster than CPython. This is due to the JIT compilation advantages and optimizations we discussed in previous work [4, 5], the main improvement for these concrete benchmarks comes from the allocation removal/unboxing optimization.

The geometric mean of the speedup of loop peeling is 70%, which makes benchmark times comparable with native-compiled C code. We attribute the performance gap to C code to the relative immaturity of RPython's JIT machine code backend and the naive register allocator. Also, in case of nested loops, operations are only moved out of the innermost loop. That is an issue when the innermost loop is short and a significant amount of time is spent in the outer loops. This is the case with for example SparseMatMult.

The large input parameters of the SciMark benchmarks are chosen in such a way to make the problem not fit into the CPU cache. This explains why PyPy is doing relatively better on them. The cache miss penalties are large relative to the time needed to perform the actual computations, which hides problems of the less efficient code generated by PyPy.

The speedups that LuaJIT gains from the loop optimization pass are similar to those PyPy gains. In general, LuaJIT is even closer to C performance, sometimes even surpassing it. LuaJIT is generating

[6] `http://math.nist.gov/scimark2/`

[7] `http://luajit.org/download/scimark.lua`

machine code of higher quality because it has more optimizations[8] and produces much better machine code than PyPy.

The performance of sqrt(Fix16) compared to the C version gives an indication of the overhead of using a custom class with operator overloading for arithmetic. For CPython the overhead over C is a lot larger than that of sqrt(int). In LuaJIT, the overhead is very small. For PyPy, sqrt(Fix16) 2.2 times slower than the C version. However, that is not actually due to the overhead of operator overloading but due to the additional overflow checking necessary for integer arithmetic in Python. The JIT does not manage to prove that the integer operations in these benchmarks cannot overflow and therefore cannot optimize away the overflow checking. This is also the reason why sqrt(float) is so much faster than sqrt(int) for PyPy. The fact that LuaJIT and PyPy do so well on sqrt(Fix16) shows that the allocation removal/sinking optimizations work well in both JITs.

8. Related Work

Loop invariant code motion optimizations are a well-known approach to optimize loops [18]. Therefore, the effects that the optimizations described here achieve are not in any way new. However, we think that achieving them in the way described in this paper is simpler than writing explicit algorithms.

Loop invariant code motion has been part of early compilers since the 1960s [1]. A common approach for achieving loop invariant code motion is to perform partial redundancy elimination. The approach was first proposed by Morel and Renvoise [17]. It involves solving data flow problems of bidirectional data flow equations. After improvements [9, 10] this approach was followed by the work of Knoop et.al. [15] who cleanly separated the problem into a backward and forward data flow analysis. Implementing partial redundancy elimination in compilers that use SSA form [8] simplified the algorithms, because no iterative data flow analysis was needed any more.

As described in the introduction, Mike Pall pioneered the approach described in this paper. He showed that, unlike traditional loop-invariant code motion (LICM), this approach is effective, even in the presence of many guards and global control dependencies, which are caused by the semantics of dynamic languages.

He writes on the Lua-users mailing list: "The LOOP pass does synthetic unrolling of the recorded IR, combining copy-substitution with redundancy elimination to achieve code hoisting. The unrolled and copy-substituted instructions are simply fed back into the compiler pipeline, which allows reuse of all optimizations for redundancy elimination. Loop recurrences are detected on-the-fly and a minimized set of PHIs is generated." [19]

Both the Hotpath VM [14] and SPUR [3] implement loop-invariant code motion directly, by explicitly marking as loop-invariant all variables that stay the same along all looping paths and then moving all pure computation that depends only on these variables out of the loop. SPUR can also hoist loads out of the loop if nothing in the loop can ever write to the memory location. It can also move allocations out of the loop, but does not replace the object by its attributes. This saves only the allocation, not the access to the object attributes.

The type specialization described by Gal *et al.* [12] can be seen as doing a similar optimization (again by manually implementing it) as the one described in Section 6.3: The effect of both is that type checks are fully done before a loop is even entered.

9. Conclusions

In this paper we have studied loop invariant code motion during trace compilation. We claim that the loop peeling approach of LuaJIT is a very convenient solution since it fits well with other trace optimizations and does not require large changes to them. The approach improves the effect of standard optimizations such as redundant guard removal, common subexpression elimination and allocation removal. The most prominent effect is that they all become loop invariant code motion optimizations.

By using several benchmarks we show that the proposed algorithm can significantly improve the run time of small loops containing numerical calculations.

The described approach still has some limitations which we plan to address in the future. In particular loop peeling works poorly in combination with trace trees [13] or trace stitching [12]. The side exits attached to guards that fail often currently have to jump to the preamble.

[8] See `http://wiki.luajit.org/Optimizations`

Acknowledgments

We would like to thank Samuele Pedroni, Sven Hager, David Schneider, and the anonymous reviewers for helpful comments on drafts of this paper. We owe gratitude to Mike Pall for making his impressive work on LuaJIT publicly available and for detailed reviews on drafts of the paper.

References

[1] F. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, Montreal, Quebec, Canada, 2007. ACM.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*, Reno/Tahoe, Nevada, USA, 2010. ACM.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In *PEPM*, Austin, Texas, USA, 2011.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, A. Rigo, and S. Pedroni. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *ICOOOLPS*, Lancaster, UK, 2011. ACM.

[6] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*, pages 18–25, Genova, Italy, 2009. ACM.

[7] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for Prolog execution. In *PPDP*, Hagenberg, Austria, 2010. ACM.

[8] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, page 273–286, New York, NY, USA, 1997. ACM.

[9] F. C.-T. Chow. *A portable machine-independent global optimizer–design and measurements*. PhD thesis, Stanford University, Stanford, CA, USA, 1984. AAI8408268.

[10] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst.*, 13(2):291–294, Apr. 1991.

[11] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106–117, May 1998.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, PLDI '09, New York, New York, 2009. ACM. ACM ID: 1542528.

[13] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of

Information and Computer Science, University of California, Irvine, Nov. 2006.

[14] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, Ottawa, Ontario, Canada, 2006. ACM.

[15] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Not.*, 27(7):224–234, July 1992.

[16] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, page 111–120, New York, NY, USA, 2005. ACM. ACM ID: 1064996.

[17] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, Feb. 1979.

[18] S. S. Muchnick and Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Sept. 1997.

[19] M. Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities, Nov. 2009. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[20] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM.