PyJIT
Dynamic Code Generation From Runtime Data

Simon Burton
Statistical Machine Learning
National ICT Australia

June 2006

Abstract

PyJIT is a new framework for providing just-in-time compilation from within Python programs. The idea is to generate machine code at runtime, and inline our data as we go. We show that this can provide dramatic speed-ups versus compiled code, by reducing memory-bandwidth use and simplifying the executable code. At the core we use the *Low Level Virtual Machine* (LLVM) which has facilities for creating machine dependant code on the fly. PyJIT handles conversion to LLVM Static Single Assignment form, and has a compiler that accepts a subset of the Python syntax, including basic control flow. We demonstrate applications in vectorized operations and numerical linear algebra, tree structures for machine learning, and optimization techniques using interval arithmetic.

1 Introduction

Whenever an operation depends on complex but static data, it is a candidate for a *Just In Time* (JIT) compilation strategy. The idea is to hard-code (inline) this data into the executable code. Speedups then occur because the CPU is less memory bound.

Here is a pseudo-example. Suppose we have a `filter`[1] operation to apply to a stream of data. The filter has some parameters, but they stay constant over the whole stream:

```
for item in stream:
  filter( parameters, item )
```

Instead of this, once the parameters become known we push them into the filter operation by generating a function `filter_1`, at runtime, with these values hardcoded:

[1] not to be confused with the Python builtin

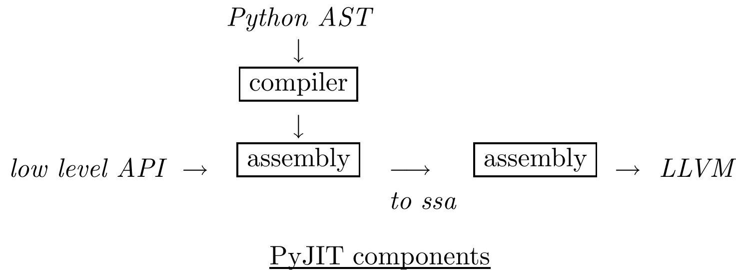
```
for item in stream:
  filter_1( item )
```

This kind of dynamic inlining is not possible with compiled code, because once code has been compiled the set of functions is fixed.

The inlining can be explicit or implicit, and there are several possibilities to consider. If the data represents control flow we can generate corresponding control flow directly. If the data contains values that form part of an operation we can inline those values into the operations themselves. There may be redundancy in a calculation, or a caching strategy that dictates how to traverse memory, that is impossible to predict (pre-code) until the data arrives.

The *Low Level Virtual Machine* (LLVM) is a collection of C++ libraries designed to be used in the back end of compilers. It is able to produce optimized machine code instructions from an assembly language. This assembly can be either directly parsed (as text) by LLVM or programmatically constructed using LLVM's C++ interface.

At the lowest level, PyJIT exposes a number of Python classes that map directly to syntactic elements of LLVM assembly. There is also a rewriting mechanism for converting those elements to the *Static Single Assignment* (SSA) form which LLVM expects. The third ingredient is a compiler that supports a small subset of the Python language, including basic control flow.



PyJIT components

There are two ways of driving PyJIT: either from Python source code (via an AST) or by constructing the assembly objects with a low level API.

1.1 Overview of Applications

The "holy grail" of many efforts is to speed up Python execution. In the first application we present, we treat Python source code as the static data. Here the user is the (runtime) programmer and we seek a way to "inline" the source itself into executable code.

In the second application we take a filter operation from numerical linear algebra, similar to the pseudo-example above.

The third application is from machine learning. This data, *decision trees*, encodes control flow. Each node in the tree is an *if* statement of some sort, and we can generate corresponding branch statements in the assembly.

The final example involves evaluating large mathematical expressions. This time the control flow is trivial: we "unroll" the tree (bottom-up) into a sequence of arithmetic operations. In this case the PyJIT approach yields performance which is comparable with a compiled, stack based evaluation engine.

2 The PyJIT Compiler

Here we present an overview of the PyJIT compiler component. We will look at examples of its use in the applications below.

The PyJIT compiler component generates assembly. As input it accepts a subset of the Python syntax. It is essentially piggy-backing onto the Python syntax; the goal is not to translate whole Python programs but to be able to write simple numeric processing functions in a higher level language. Surprisingly, this turns out to be quite useful.

The compiler works by visiting the *Abstract Syntax Tree* (AST) of Python functions. There are several passes. One pass generates the units of control flow (the *Basic Blocks*, see below). Another pass propagates type information, starting with the function signature.

Sequence indexing operations are treated as C-array like operations, and are translated into the corresponding memory load or store operation. The compiler implements pointer arithmetic (adding or subtracting an offset from a pointer variable), and `struct` member access via the usual attribute syntax.

All these operations are reasonably easy to translate into the assembly language. But, generating correct assembly for non-trivial control flow is rather more tricky. Thus, one useful aspect of the compiler is its ability to generate code for `if` statements and `while` loops.

3 The Low Level Virtual Machine

The LLVM assembly is a platform independant assembly language [2]. There is no notion of registers; variables can be created freely.

The type system covers most of the types from the c-level: basic integer and floating point types, as well as typed pointers, structs, arrays, and packed types (used in SIMD operations).

Functions are built from a sequence of *basic blocks*. Each block contains a sequence of instructions, and ends with a terminator instruction directing control flow to either another basic block or to return from the function.

3.1 Static Single Assignment Form

LLVM assembly is required to be in *Static Single Assignment* (SSA) form, which means that each variable is assigned to (appears as an lvalue) only once. In practice this can be done by "versioning": we decorate the variable's name with a version number. Each time a we assign to that variable, we increment the version:

```
i0 = 4
i1 = i0 + 16
i2 = i1 + 16
```

The advantage here is that we can easily deduce the flow of data: at every location where a variable is used we can find the instruction where that data came from, because it was assigned to exactly once.

A moment's reflection reveals the difficulty with this: how do we compute with a variable that could have travelled many paths to get here ?

```
if test:
  i = 1
else:
  i = 0
j = i + 1
```

Or how does one build a while loop, since we need to both initialize a counter and then increment it, which requires two assignments to the same variable?

```
def count(n):
  i = 0
  while i < n:
    i = i + 1
```

What we are trying to do is fold multiple values into one value. The place where this folding occurs, where control flow is converging, is called a *join point*. SSA form has a special instruction called a ϕ-*instruction* which is placed at the joins. These are placed at the beginning of a basic block and list possible values along with the basic block each value came from.

So, each time we assign to a variable, we give it a new name, and whenever we need to merge two variables (from different histories) we use a ϕ-instruction.

Here is assembly for the `while` loop. Comments come after the semi-colon, and the variables are indicated with the `%` sign:

```
void %count(int %n) {
  bb.0:                           ; BASIC BLOCK 0
    br label %bb.1                ; BRANCH
  bb.1:                           ; BASIC BLOCK 1
    %i.0 = phi int [ %i.1, %bb.2 ], [ 0, %bb.0 ]
                                  ; PHI
    %var.0 = setlt int %i.0, %n   ; var.0 = i.0 < n
    br bool %var.0, label %bb.2, label %bb.3
                                  ; IF BRANCH
  bb.2:                           ; BASIC BLOCK 2
    %i.1 = add int %i.0, 1        ; i.1 = i.0 + 1
    br label %bb.1                ; BRANCH
```

```
  bb.3:                              ; BASIC BLOCK 3
    ret void                         ; RETURN
}
```

The problem of finding where to insert ϕ-instructions is non-trivial. For each basic block assigning to some variable it involves finding the first successors that may see other values for that variable (coming from other basic blocks). Such successors are known as the *dominance frontier* for that basic block [4].

The SSA form enables many compiler optimizations. PyJIT handles *Constant propagation*. LLVM is able to perform *Dead code elimination*, *Strength reduction* and *Register allocation*. SSA also assists in *Code re-ordering*, *Parallelization* and *Partial redundancy elimination*.

4 Applications

For the following benchmarks and timings we use a dual core 3.2GHz Intel Pentium 4 machine with 800MHz FSB, and 1GB 400MHz DDR SDRAM Memory.

Times are real times (wall clock times), averaged over many repetitions, except for the CG operations which were only performed once.

Software used: LLVM version 1.7, compiled without assertions or debug info. gcc version 3.4.1. debian gnu-linux, sarge, with libc SSE bug-fix. NumPy version 0.9.9.2537 (SVN). Python 2.4.3. NumExpr revision 1762. Psyco-1.5 revision 27794.

4.1 Vector Operations

Numpy is a Python library for the manipulation of arrays. The array elements must be of a fixed size, eg., 32 bit int's.

The trick with writing high performance code in NumPy is to "vectorize" each operation. Instead of looping over an array and performing operations on each element (too expensive at the Python level), one decomposes the operation into primatives that are implemented inside of NumPy as a c-code for loop.

This example takes two arrays: **values**, and **weights**. It sums elements of the **weights** array whose corresponding element in the **values** array is below a given **cutoff**. Some values are regarded as unknown, these are stored as NaN's and do not contribute to the final weight.

To implement this in NumPy, we first record the result of the filter operation in a boolean mask array, then build an integer array containing the indices of the nonzero elements of the mask, use those indices to select a subarray of the original **weights** array, and then finally we sum those elements:

```
def get_weight( cutoff, values, weights ):
  k_mask = -numpy.isnan(values)       # mask of known values
  lt_mask = (values<cutoff) & k_mask
  lt_indices = numpy.nonzero(lt_mask) # array of indices
  lt_weights = weights[ lt_indices ]  # a new array
```

```
    return numpy.sum( lt_weights )
```

The equivalent Python code using a while loop is more straightforward but for large arrays it is around 200 times slower.

```
def get_weight( cutoff, values, weights, N ):
  i = 0
  weight = 0.0
  while i < N:
    value = values[i]
    if isnan( value )!=0 and value<cutoff:
      weight = weight + weights[i]
    i = i + 1
  return weight
```

However, this is now in a form where we can use PyJIT to compile it to LLVM assembly:

```
ll = compile(get_weight,
  [ llasm.double, llasm.double.pointer,
    llasm.double.pointer, llasm.int_],
  isnan=llasm.int_ )
```

The **compile** function takes as arguments the Python function to compile, the signature of the function, and any keyword arguments are interpreted as types of variables: in this case we need to state the return type of **isnan** (a standard c library function).

For the benchmarks we use uniform random arrays for **weights** and **values**, 5% NaN's, and a cutoff that selects half the weights for summing. `N` is the size of the input vectors (arrays).

N	Python	Psyco	NumPy	PyJIT	final speedup
$1e3$	6.6mS	3.5mS	160μS	34μS	x4.7
$1e4$	44mS	31mS	700μS	270μS	x2.6
$1e5$	430mS	300mS	7.2mS	2.8mS	x2.5
$1e6$	4.2S	3.0S	71mS	27mS	x2.7

Timings for **get_weights** function

The essential difficulty with the NumPy version is that it creates and traverses big temporary arrays: four different mask arrays, each with N elements, an array of indices, and a final temporary array to sum over. After each step of the calculation the array result has (mostly) left the cache: performance degrades as we need to use memory bandwidth to reload this array.

NumExpr [2] is able to combat this cache abuse by performing the calculations blockwise on the arrays. It uses a small virtual machine, written in C, that is driven by a bytecode. The bytecode is compiled from python expressions.

Psyco is a JIT for python. It works by observing variables at runtime and generating machine code operations for any native types. However, performance suffers when it comes across an operation that cannot be handled natively: then it must call back into the CPython interpreter. In the previous case calling the `isnan` function triggered this behaviour.

In order to favourably compare these tools, for the next benchmark we do a simpler calculation. The NumPy `where` function selects elements from two arrays based on a boolean condition array. To get maximum performance out of Psyco we used the python `array` module to store the arrays.

N	Python	NumPy	Psyco	NumExpr	PyJIT
$1e3$	490μS	100μS	45μS	27μS	20μS
$1e4$	5.0mS	780μS	430μS	250μS	120μS
$1e5$	51mS	9.4mS	4.4mS	3.5mS	1.4mS
$1e6$	510mS	94mS	44mS	35mS	13mS

Timings for `where(a<cutoff,b,c)`

4.2 Numerical Linear Algebra

PETSc, the *Portable, Extensible Toolkit for Scientific Computation*, is a library of C routines revolving around linear algebra components. It can handle sparse and dense matrices, either distributed across several computers for parallel computation, or existing on a single computer for a sequential computation.

Matrix free methods refer to the ability for PETSc to use "abstract"(shell) matrices that are not necessarily based on an array of data: the user provides callbacks for performing operations using this matrix, such as matrix vector multiplication. Other PETSc components, such as the linear solvers, can then be used with these shell matrices.

We examine the problem of solving a linear system

$$Ax = b$$

For A we use an n by n tridiagonal matrix with 2 on the diagonal and -1 on the off diagonals:

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & \\ -1 & 2 & -1 & 0 & 0 & \cdots \\ 0 & -1 & 2 & -1 & 0 & \\ \vdots & & & & & \ddots \end{bmatrix}$$

[2] by David M. Cooke and Tim Hochberg

And the rhs vector, $b = [1, 0, \cdots, 0, 1]^t$.

We compare the performance of using a PETSc sparse AIJ matrix (these use compressed sparse row storage) with a shell matrix that has its matrix vector multiply operation implemented as a PyJIT function:

n	AIJ time	PyJIT time	speedup
$1e4$	480μS	75μS	x6.3
$1e5$	4980μS	373μS	x13
$1e6$	46mS	290μS	x16

Timings for Matrix Vector Multiply

Once this operation is implemented it is enough for the resulting matrix shell to be used in PETSc's linear equation solver's, in particular we test the *conjugate gradient* (CG) solver.

n	iterations	AIJ time	PyJIT time	speedup
$1e4$	5000	3.3S	1.3S	x2.6
$1e5$	10000	83S	43S	x1.9

Timings for CG Solver (no preconditioning)

The following source code implements the operation:

```
def mymult(x, y, n):
  y[n-1] = 2.0 * x[n-1] - x[n-2]
  i = n-2
  while i > 0:
    y[i] = 2.0 * x[i] - x[i-1] - x[i+1]
    i = i - 1
  y[0] = 2.0 * x[0] - x[1]
```

4.3 Decision Trees

Decision trees provide a simple approach to machine learning [3]. The data consists of a number of example cases, each case has certain attributes and an outcome label. The goal is to be able to predict the outcome of new cases based on the attributes only.

The algorithm works by recursively splitting the cases, using a test on the attributes that partition the labels the best.

For discrete attributes we can split the cases into classes that contain only that value of the attribute. For continuous attributes we choose a threshold and split the cases into those where this attribute is lower than the threshold and those where this attribute is greater than the threshold.

For example, we might try to predict if it will rain based on other attributes such as "is the sun shining?", temperature and humidity. Here we have two

kinds of attributes: “is the sun shining?” has a discrete value, either “yes” or “no”, and the other attributes have continuous values. The example cases will come from past history:

Sunny	Temp	Humidity	Rain ?
yes	69	70	no
yes	72	95	no
yes	75	70	no
yes	80	90	no
yes	85	85	no
no	64	65	no
no	65	70	yes
no	68	80	yes
no	70	96	yes
no	71	80	yes
no	72	90	no
no	75	80	yes
no	81	75	no
no	83	78	no

What we see straight away is that if we split on the “Sunny” attribute, the “yes” partition is entirely “no rain”. Then, for the “not sunny” partition if we further split on the temperature attribute with a cutoff at 71.5; those with lower temperature predict as “yes rain” and those with greater temperature predict as “no rain” we see that we correctly predict 7 of the given cases and make an error on another 2 cases:

```
if Sunny == "yes":
    Rain = "no" # 5 correct
elif Sunny == "no":
    if Temp < 71.5:
            Rain = "yes" # 4 correct, 1 error
    elif Temp >= 71.5:
            Rain = "no" # 3 correct, 1 error
```

At this point there are no further subdivisions which will improve the error rate, so construction terminates.

The tree construction boils down to choosing the best splitting criterion at each level of the tree. This is a brute force search for a maximum, and there is only a little data that can be set constant during the search. Therefore the PyJIT approach does no better than compiled code. However, evaluating the tree over a test set of cases is where a jit strategy provides big wins. We use PyJIT to construct the decision tree directly in terms of control flow instructions.

Here is the assembly, with memory access operations removed for clarity:

```
bb.1:
  switch int %Sunny, label %bb.0
    [int 0, label %bb.2  int 1, label %bb.3]; Sunny = 0 or 1
bb.2:
  ret int 0; no rain
bb.3:
  %result = setlt double %Temp, 71.5; temp < 71.5 ?
  br bool %result, label %bb.4, label %bb.5
bb.4:
  ret int 1; rain
bb.5:
  ret int 0; no rain
bb.0:
  ret int -1; error
```

This was compared with a straight forward implementation in C. It is a non-recursive for-loop that traverses the tree to the leaves. The nodes of the tree are 24-byte structs, allocated in an arena to improve cache performance. The C version has to traverse the tree data structure as well as evaluating the test data and so ends up being slower than the PyJIT version which only needs to read the test data. We also compare with a Python version that uses NumPy.

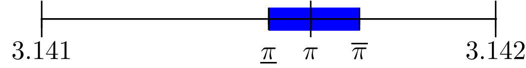
For the following benchmark we use synthetic binary trees, and an ordered data set that exercises the paths to the leaves. We have two kinds of tree, *easy* and *hard*. In the hard case, the data is fed to the tree in a way that maximizes cache misses: each traversal of the tree uses code (control flow) that has the least in common with the previous traversal. Conversely, the easy case minimizes cache misses: the control flow changes the least possible between data items.

tree size	tree shape	code size	codegen time	Python time	C time	PyJIT time	final speedup
511	easy	1580B	0.35S	730mS	18mS	6.7mS	x2.8
511	hard	1580B	0.37S	730mS	20mS	7.7mS	x2.6
2047	easy	6188B	0.80S	1.1S	27mS	11mS	x2.6
2047	hard	6188B	0.81S	1.2S	36mS	21mS	x1.7
8191	easy	24620B	2.7S	2.8S	34mS	15mS	x2.3
8191	hard	24620B	2.7S	2.6S	49mS	29mS	x1.7

Decision tree benchmarks

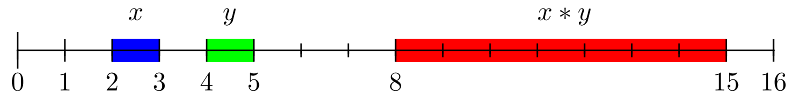
4.4 Interval Arithmetic

Interval Arithmetic is based on two ideas. First, instead of using higher and higher precision to represent real numbers, we store lower and upper bounds (an *interval*) that are represented exactly using machine floating point numbers.



We don't know what the number is exactly but we know it is inside the interval somewhere (maybe at one of the endpoints).

The second idea is that we carry out all our arithmetic operations on these intervals such that the resulting interval encloses all possible values:

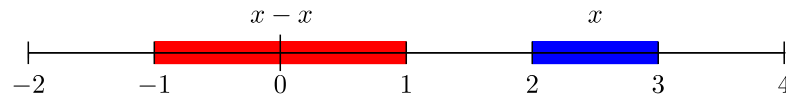


We enlarge the result if necessary so that the endpoints are represented exactly in the machine.

Calculations involving intervals amount to a (computer generated) mathematical proof: it *guarantees* that the result of the calculation is inside an interval [1].

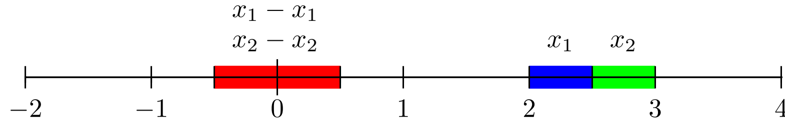
Unfortunately, this guarantee can sometimes become extremely weak, as the interval explodes. What happens is that as the calculation progresses it loses information about the corellations between the values in the calculation.

A simple example is given by subtracting an interval from itself. The "real" answer is zero; a very small interval, but we end up with something considerably larger:



Methods exist to counter this effect by augmenting the interval object with information describing how its value depends on another value; for example, derivative information.

This method carried to its extreme leads to the following idea. We treat a calculation *lazily*, ie., as an interval valued function of intervals. Then, to find a tight interval enclosing the result we use a minimization/maximization algorithm which has been extended to handle interval values. For example, as we apply a bisection algorithm to the calculation $f(x) = x - x$ we see that the resulting interval approaches zero.



The implementation of this requires that function evaluations be as fast as possible.

We compared a version using PyJIT with compiled code. The compiled code used arenas containing the tree; each node has a function pointer and child pointers, each leaf records an index to an interval value. The code then traverses this tree using an instruction stack and an intermediate value stack. The PyJIT version carries out the same operations except it can completely "unroll" the tree traversal.

The two approaches yield similar performance, however the author cannot escape the feeling that writing the code for emitting PyJIT constructs was a lot easier than writing a stack based interpreter in c.

5 Related Work and Future Directions

The PyPy project was the main inspiration for this work. They are able to translate a significant portion of the Python language (RPython) which includes all of the Python object semantics, and also provide JIT related facilities. PyJIT is specifically targeted at either generating low-level assembly code that does not correspond to any source code, or using a small portion of the Python syntax and semantics to drive compilation to assembly.

The *Python Specializing Compiler* (Psyco) is an extremely useful tool for speeding up Python programs. It's Python coverage is fairly complete, in that the code will call back into CPython whenever it encounters an operation that cannot be handled natively. The tradeoff is that whenever this happens it is slow again. Also there is no easy way to extend the system; and the only way to generate functions dynamically is via python source code.

There are other compilers such as *Shed Skin* and *Pyrex* but these rely on a full C/C++ compiler and are not really targeted at dynamic code generation. Pyrex needs an augmented syntax, essentially c-code mixed with python code. ShedSkin's Python coverage is incomplete.

From here, it is a short hop for PyJIT to being able to translate all of the Python semantics into assembly, similar to Pyrex and Psyco. However, one killer application will be translating NumPy semantics into optimized code, as was done by hand in section 4.1. It is also clear that the PyPy project has significant overlap with the present effort, and this needs to be investigated further.

6 Conclusion

PyJIT uses LLVM to generate extremely fast code on the fly.

This technique forces us to think differently about algorithm design. The biggest wins come when static contextual data is large compared to the dynamic data, and dictates some aspect of the control flow. We then get big improvements in performance by inlining this data into the executable code.

References

[1] R. Hammer, M. Hocks, Ulrich Kulisch, and Dietmar Ratz. *Numerical Toolbox for Verified Computing I, Basic Numerical Algorithms, Theory, Algorithms, and Pascal-XSC Programs.* Number 21 in Springer Series in Computational Mathematics. Springer-Verlag, New York, 1991.

[2] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[3] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1992.

[4] Mark N. Wegman and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct 1991.