Vectorization in PyPy's Tracing Just-In-Time Compiler

Richard Plangger
Andreas Krall

TU Vienna - Institut für Computersprachen

Mai 24, 2016

Outline

Meta-Interpreter
An approach to VM construction

Vectorization Algorithm
High level view to some important details

Embedding it into a TJIT
Details about the implementation

Benchmark Results

Meta-Interpreter

pypy

A bird's eye view

1. Virtual machine for Python
2. Tracing JIT compiler
3. Moving gnerational GC (Mark and sweep, incremental)
4. Extensible/modular architecture

But we **did not** build a TJIT/GC for Python

JITs are often strongly tied to interpreter & language internals

Components, newly invented over and over

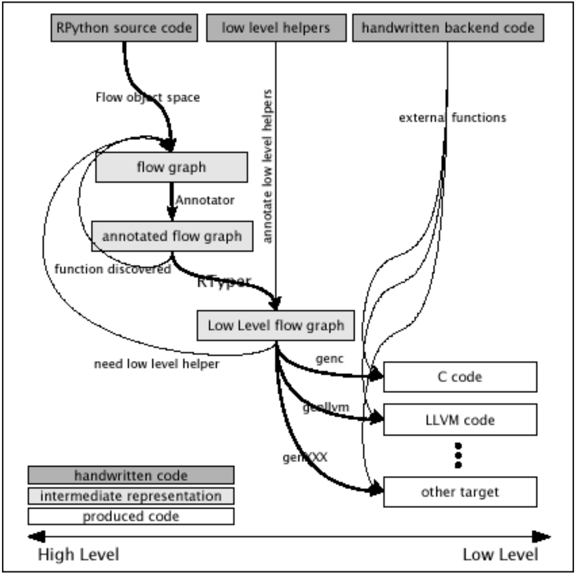
- Bytecode/AST to IR
- Optimizer
- Register allocator
- Code generation

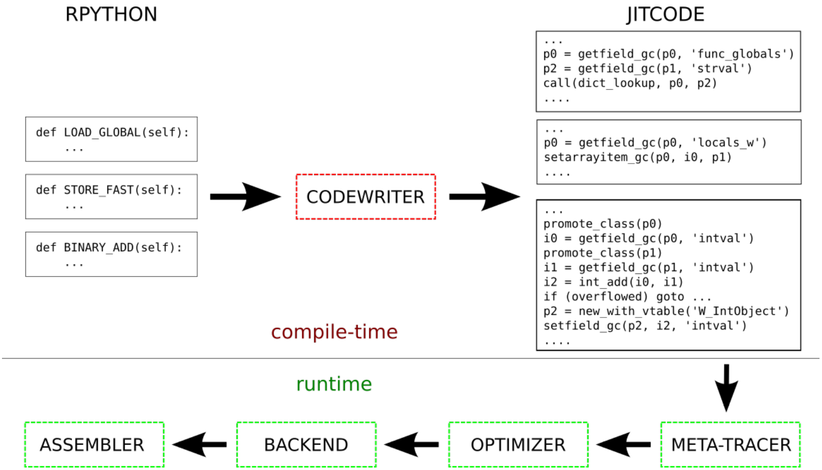
Those are (amongst others) tricky to get right and require a lot of work

RPython

High level language to aid VM construction

1. Import the complete program
   Initialisation can use full Python
2. Process the code object (abstract interpretation)
   Yields control flow and data flow
3. **Annotation**
   Deduce types starting from the interp. entry point
4. **RTyping**
   Converts graphs to low level operations
5. **Codegeneration**
   Emits C code that is later compiled


RPython source code
low level helpers
handwritten backend code
Flow object space
annotate low level helpers
external functions
flow graph
Annotator
annotated flow graph
function discovered
RTyper
Low Level flow graph
need low level helper
genc
C code
genllvm
LLVM code
genXXX
other target
handwritten code
intermediate representation
produced code
High Level
Low Level


RPYTHON
JITCODE
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
....
def LOAD_GLOBAL(self):
...
...
p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....
def STORE_FAST(self):
...
CODEWRITER
...
promote_class(p0)
i0 = getfield_gc(p0, 'intval')
promote_class(p1)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(i0, i1)
if (overflowed) goto ...
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
....
def BINARY_ADD(self):
...
compile-time
runtime
ASSEMBLER
BACKEND
OPTIMIZER
META-TRACER

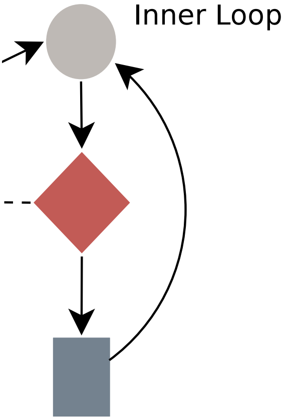
Terminology

- **Translation:** Transforming an interpreter to an executable
- **Tracer:** Attached to the interpreter, records it’s steps
- **Jit Code:** Code the tracer executes to record steps
- **Trace:** Linear sequence of instructions
  Single entry, multi exit
- **Guard:** Instruction to ensure correctness
  Bails out of the trace if it fails
- **Bridge:** A trace that is attached to a guard
  Attaching a bridge is also called “stitching”

Tracing JIT

Trace: List of instructions (Single entry, multi exit)

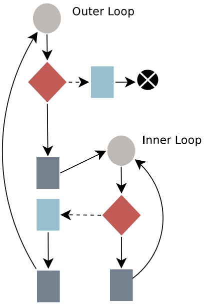
```
# Inner Loop
i = 0
while i < X:
    x = func(i * 33)
    if x != 0:
        break
    p[i].x = x // 2
    i += 1
```



Tracing JIT (II)

Procedure of building trace tree is “recursive”

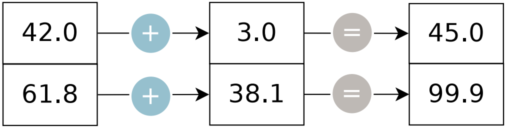
- JitDriver used to be able to trace a dispatch loop
- Entering and leaving another JitDriver supported
- Function tracing



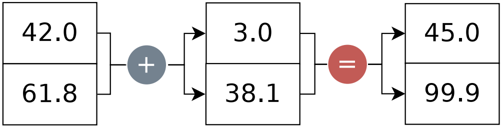
Vectorization Algorithm

Superword parallelism

Element wise addition of two vectors



Single Instruction Multiple Data



Hardware supported (e.g. SSE, NEON, ...)

Motivation

NumPy, a versatile array processing library

? *Why does NumPy on PyPy not work out of the box*

GC Scheme + C level API of CPython

Solution was needed to optimize the array processing

Potential to use it in regular Python programs

Vectorization

Contradicting goals in a JIT compiler

1. Time & space requirements
2. Traces instead of regions
3. Specifically targeting SIMD instructions

Traces might contain enough parallelism

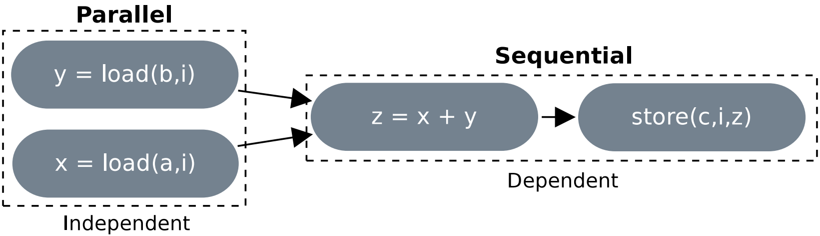
Building blocks

1. Loop unrolling
   if necessary
2. Data structures to express dependencies
3. Analysis step
4. Transformation
5. Adapted code generation

Data Dependency

Graph representation of instruction dependencies.

- True dependency
- Anti dependency
- Output dependency
- (Control dependency)



Ubiquitous guard exists

Introduces many dependencies

Need to be eliminated, no validity preserving transformation possible

Code motion moves guard as early as possible

Move a path of pure operations earlier

Instruction Parallelism

Analysis step to group instructions

A simple greedy comparison of operations initiated by load/store operations

- $O(n^2)$ to compare each instruction with another
  n ... # trace instructions
- Only load/store instructions are considered at first
- Extension phase follows dependencies
  To reveal other parallel instructions

Grouping of “isomorphic” instructions

Same IR opcode and argument types

Instruction Parallelism (II)

Resulting information contains

1. **Pairs:** Tuple of parallel isomorphic instructions
2. **Pack:** N-Tuple of parallel isomorphic instructions

Transformation pass

Can be done by re-scheduling the trace considering pairs and packs

Code generation

Scheduling

Work through the dependency graph:

1. Pick, remove and emit a schedulable node
2. Remove edges and recompute the set of schedulable nodes

Pack dependencies

Cycle can only be broken by partly/fully removing the pack restriction

Scheduling II

Additional enhancements done while scheduling:f

1. Vector cropping. Size of the input vector is too big/small
   Integer sign extensions
2. Vector slot movement
   Conversion 32-bit float to 64-bit float
3. Invariant scalar/constant expansion
4. Inline scalar/constant expansion

Example

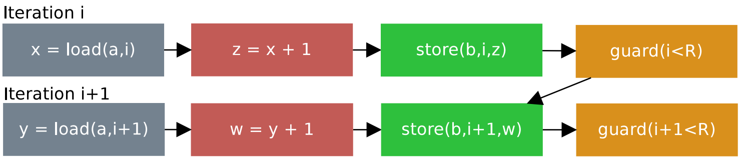
```
i = 0
while i < R:
    b[i] = a[i] + 1
    i = i + 1
```

```
x = load(a,i)
z = x + 1
store(b,i,z)
guard(i+1 < R)
# iteration i+1
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
guard(i+2 < R)
```

Example

```
i = 0
while i < R:
      b[i] = a[i] + 1
      i = i + 1
```

```
x = load(a,i)
z = x + 1
store(b,i,z)
guard(i+1 < R)
# iteration i+1
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
guard(i+2 < R)
```



Tracing complications

```
x = load(a,i)
z = x + 1
store(b,i,z) ●
guard(i+1 < R) ●
y = load(a,i+1)
w = y + 1
store(b,i+1,w) ●
guard(i+2 < R)
```

? *Store operations independent*

Counter example: Guard fails, but store(b,i+1,w) already executed.

Valid to execute guard earlier

“Guard Early Exit”

```
guard(i+1 < R)
guard(i+2 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
# guard(i+1 < R)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
# guard(i+2 < R)
```

Move guards to an earlier place. Scheduling reorders instructions.

“Guard Early Exit”

```
guard(i+1 < R)
guard(i+2 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
# guard(i+1 < R)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
# guard(i+2 < R)
```

Move guards to an earlier place. Scheduling reorders instructions.

! *Pure operations must precede guards*

```
guard(i+1 < R)
guard(i+2 < R)
guard(i+3 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
v = load(a,i+2)
q = v + 1
store(b,i+2,q)
```

```
guard(i+1 < R)
guard(i+2 < R)
guard(i+3 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
v = load(a,i+2)
q = v + 1
store(b,i+2,q)
```

```
guard(i+1 < R)
guard(i+2 < R)
guard(i+3 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
v = load(a,i+2)
q = v + 1
store(b,i+2,q)
```

Packing

```
guard(i+1 < R)
guard(i+2 < R)
guard(i+3 < R)
x = load(a,i)
z = x + 1
store(b,i,z)
y = load(a,i+1)
w = y + 1
store(b,i+1,w)
v = load(a,i+2)
q = v + 1
store(b,i+2,q)
```

Packs are a representation of vector instructions

1. Independent instructions
2. Isomorphic instruction pairs/packs

Vector loop

```
label(a,b,i,R)
guard(i+3 < R)
[x,y,v] = vec_load(a,i)
[z,w,q] = [x,y,v] + [1,1,1]
vec_store(b,i,[z,w,q])
jump(a,b,i+3,R)
```

Embedding it into PyPy

Embedding it into PyPy

Optimization just before backend assembly

- Just after the “unrolling optimization”
  Guard strength reduction, invariant code motion, object virtualization

Roughly 4000 lines of code
+ 4000 for testing

Accumulation

Reduction cannot be represented

Need to carry information out of trace loops and recognize the pattern

Chained computations can be matched, saved as accumulation pack

1. Use an accumulation vector to save the computation
   In each slot only a part of the information is stored
2. Several points need the resulting value
   "Flush" the real value (e.g. sum: horizontal add)
3. Scheduling pass needs to be adapted slightly

Speculative ABC optimization

Array bound checks are not fully eliminated

Loop bounds and array bounds are checked

Speculative step to remove guarding instructions.

If the loop bound is smaller than the length of the array, no IndexError cannot occur on that array.

Transitive relation introduced that is checked before the vector loop

Version trace loops

? *Switch back to interpreter always necessary?*

Several iterations needed to complete the loop (odd vector length)

1. Directly attach versions of the loop to the loop exit
2. As well as to guards for ABC

No need to switch back to the interpreter

Extensions

The following has been added:

- Constant/Scalar expansion
- Accumulation
- Speculative ABC optimization for array accesses
- Trace Loop versioning

Future work

- Aligned memory access not yet supported
- No reordering support of interleaved formats

Evaluation

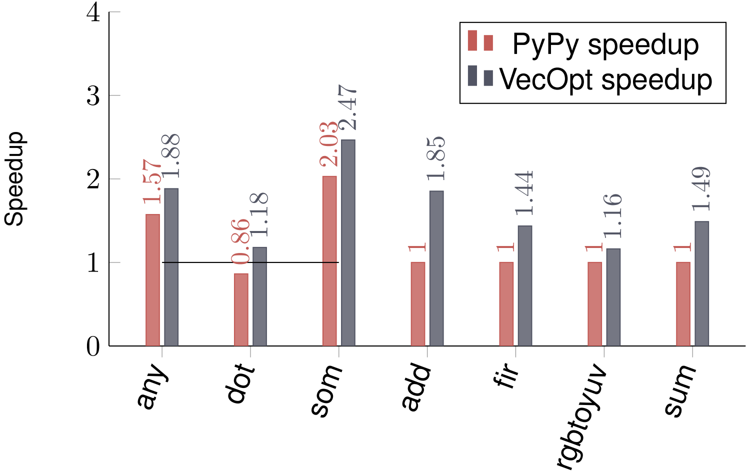
Optimization time

Count	Instruction count	Unroll factor	Microseconds
6	12-16	2	101.47
5	17-19	4	158.46
2	17	8	224.03
2	17	16	396.60

Benchmark programs

Name	CPython	PyPy	VecOpt	VecOpt Speedup
arc-distance	0.07898	0.1813	0.1608	**1.1**
diffusion	0.5603	5.665	3.889	**1.5**
evolve	0.1967	1.815	1.728	**1.1**
fft	0.9507	0.2981	0.2955	1.0
harris	0.3485	3.119	1.504	**2.1**
l2norm	0.564	1.73	1.634	**1.1**
lstsqr	0.3844	1.506	1.39	**1.1**
multiple-sum	0.1432	0.6341	0.25	**1.1**
rosen	0.5795	3.498	3.438	1.0
specialconvolve	0.4713	3.876	2.649	**1.5**
vibr-energy	0.2784	0.7552	0.699	**1.1**
wave	2.191	1.114	1.166	0.9
wdist	2.927	1.202	1.179	1.0

Python programs



Questions?