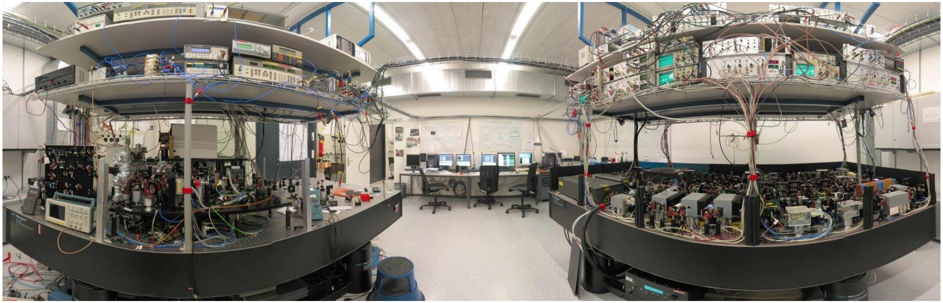
Bootstrapping a Real-Time Software Infrastructure for Quantum Physics

Robert Jördens

M-Labs, `http://www.m-labs.hk`, `mailto:rj@m-labs.hk`



M-Labs

- Founded in 2014
- Incorporated in Hong Kong
- Now 4 full-time staff

QUARTIQ

- Founded in 2017
- Incorporated in Berlin

Python

Touched Python 1.4 in 1997

Python

Stuck with it through university, Ph.D. and PostDoc

Python

Debian Developer with Python focus

Python

At home in Python-friendly quantum[a] communities

[a]simulation, communication, computation, sensing...

Python

Wrote heaps of physicist code in Python

Python

Not proud of most of them

Compiling Python?

(A) "Prototype in Python, then optimize for speed"

(A) "Prototype in Python, then optimize for speed"

- Algorithms, numerics, data wrestling, concurrency

(A) "Prototype in Python, then optimize for speed"

- Algorithms, numerics, data wrestling, concurrency
- Use JITs for CPUs/GPUs: numba, theano, numexpr, tensorflow...

(A) "Prototype in Python, then optimize for speed"

- Algorithms, numerics, data wrestling, concurrency
- Use JITs for CPUs/GPUs: numba, theano, numexpr, tensorflow...
- Refactor/rewrite as extension modules: cython, theano, C/C++...

(A) "Prototype in Python, then optimize for speed"

- Algorithms, numerics, data wrestling, concurrency
- Use JITs for CPUs/GPUs: numba, theano, numexpr, tensorflow...
- Refactor/rewrite as extension modules: cython, theano, C/C++...
- Transpile: grumpy (Go)...

(A) "Prototype in Python, then optimize for speed"

- Algorithms, numerics, data wrestling, concurrency
- Use JITs for CPUs/GPUs: numba, theano, numexpr, tensorflow...
- Refactor/rewrite as extension modules: cython, theano, C/C++...
- Transpile: grumpy (Go)...
- Swap runtime: pypy, jython, grumpy

Numba

```python
@jit('f8(f8[:])')
def sum1d(a):
    total = 0.0
    for i in range(a.shape[0]):
        total += a[i]
    return total
```

- Parse Python, Infer types, transform
- Emit LLVM IR, compile, dlopen, call
- Good for numerics, GPGPU, and data wrestling

Cython

```
cimport numpy as np
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef sum1d(np.ndarray[double, ndim=1] a):
    cdef double total = 0.0
    for i in range(a.shape[0]):
        total += a[i]
    return total
```

- Parse Cython (similar to Python), analyze, transform
- Emit C, compile, dlopen, call
- Good for numerics, extensions, wrappers

Theano

```
import theano

a = theano.tensor.vector("a", dtype="float64")
f = theano.function([a], a.sum())
```

- Analyze expression graph, algebraic transformations, optimizations
- Emit C, compile, open, execute
- An optimizing CAS, good for numerics, GPGPU, machine learning
- Similar: TensorFlow (Google)

(B) "Have code in X, want to call from Python"

(B) "Have code in X, want to call from Python"

- X $\in$ {C, C++, Fortran, Root... or even some proprietary SDK}

(B) "Have code in X, want to call from Python"

- X $\in$ {C, C++, Fortran, Root... or even some proprietary SDK}
- Cython, C/C++, cffi, ctypes, cppyy, boost.python, swig, pyROOT...

(B) "Have code in X, want to call from Python"

- X $\in$ {C, C++, Fortran, Root... or even some proprietary SDK}
- Cython, C/C++, cffi, ctypes, cppyy, boost.python, swig, pyROOT...
- Swap runtime: jython, grumpy...
- Biggest challanges: types/signatures, GC, redundant code

(C) ”Use Python to extend X”

(C) "Use Python to extend X"

- Add a REPL to some application: embed + extension

(C) "Use Python to extend X"

- Add a REPL to some application: embed + extension
- Add scripting capabilities: embed + extension

(C) "Use Python to extend X"

- Add a REPL to some application: embed + extension
- Add scripting capabilities: embed + extension
- (JIT transpiler, swap runtime)

(D) ”Targetting special runtimes”

(D) "Targetting special runtimes"

- Python for DSLs: embedded systems, specific peripherals, special architectures

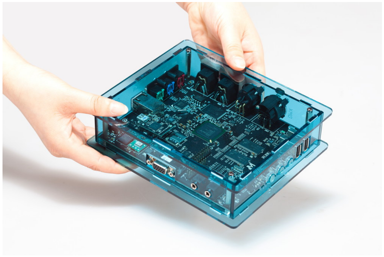
(D) ''Targetting special runtimes''

- Python for DSLs: embedded systems, specific peripherals, special architectures
- Transpile/compile/JIT: theano/numba (GPUs), ARTIQ, others

Don’t be afraid to create DSLs

Migen

History of Migen



- Built a high-level language to describe programmable logic designs efficiently (2011)
- Originally only to satisfy the need for flexible metaprogramming of complex dataflows in graphics processors.
- Found out it was excellent for SoC, started MiSoC (2012)
- Now features
  - complete powerful language,
  - large library of cores,
  - own Pythonic simulation and co-simulation toolkit, and
  - support for dozens of platforms and toolchains.
  - is broadly supported and used in many projects.

Synchronous logic

```
a = Signal()
b = Signal()
x = Signal()
module.sync += x.eq(a | b)
verilog.convert(module)
```

Synchronous logic

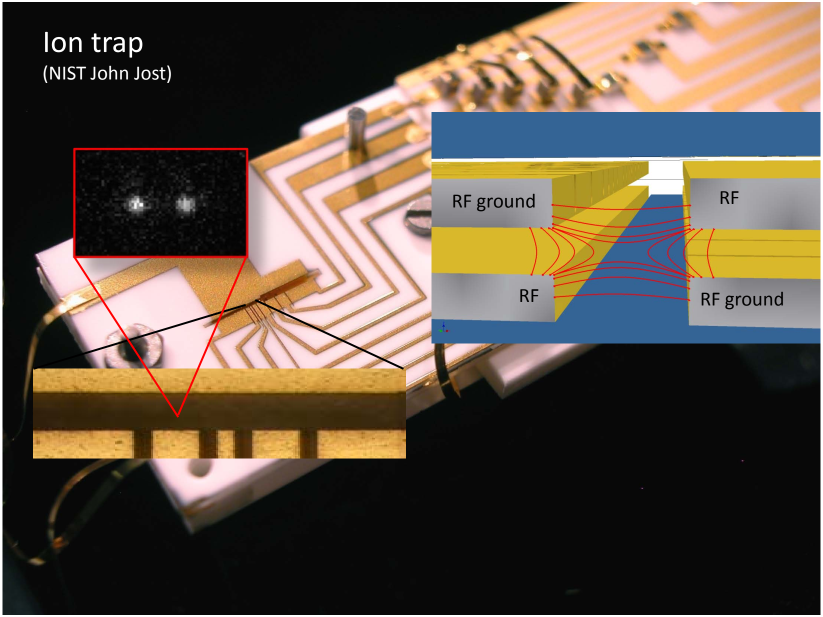
```
module top(input sys_clk, input sys_rst);

reg a = 1'd0;
reg b = 1'd0;
reg x = 1'd0;

always @(posedge sys_clk) begin
  if (sys_rst) begin
    x <= 1'd0;
  end else begin
    x <= (a | b);
  end
end

endmodule
```

Finite state machines (FSMs)

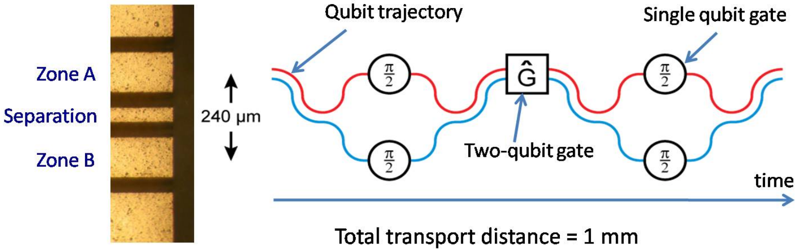
```
fsm = FSM()
fsm.act("IDLE",
    foo.eq(a & b),
    If(start_munging, NextState("MUNGING"))
)
fsm.act("MUNGING",
    foo.eq(c),
    If(back, NextState("IDLE"))
)
```

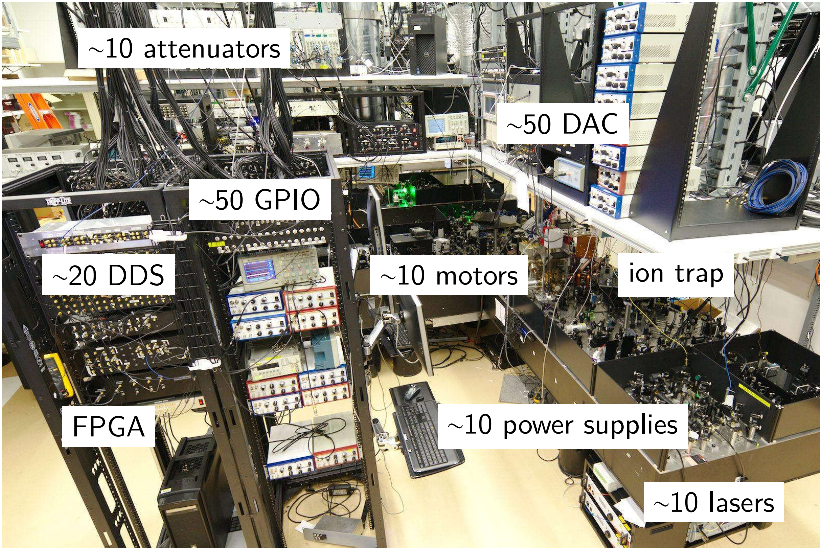
Ion trap

(NIST John Jost)



Quantum gate sequences (NIST)

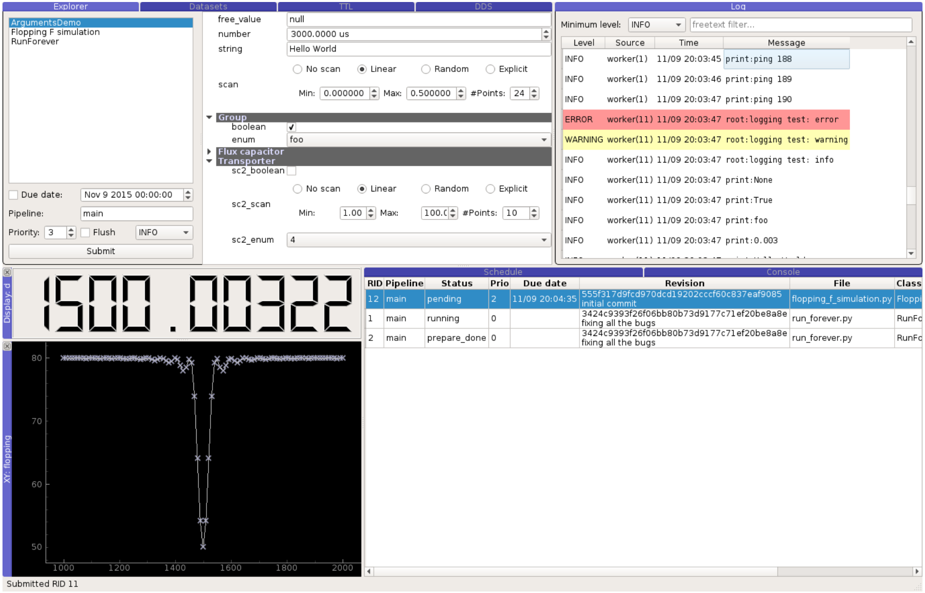



~10 attenuators
~50 DAC
~50 GPIO
~20 DDS
~10 motors
ion trap
FPGA
~10 power supplies
~10 lasers

ARTIQ

- ARTIQ is the **A**dvanced **R**eal-**T**ime **I**nfrastructure for **Q**uantum physics.
- An integrated software/gateware/hardware system that controls atomic physics experiments.
- Developed with the NIST Ion Storage Group (atomic clocks, quantum information, quantum simulation).
- Managing/scheduling experiments, driving distributed devices, analyzing/displaying/archiving results.
- Quickly and reliably deployable (Anaconda packages).

ARTIQ graphical user interface



ArgumentsDemo
Flopping F simulation
RunForever

Due date: Nov 9 2015 00:00:00
Pipeline: main
Priority: 3   Flush   INFO

free_value	null
number	3000.0000 us
string	Hello World
scan	No scan [✓ Linear] Random Explicit; Min: 0.000000 Max: 0.500000 #Points: 24
Group	
boolean	✓
enum	foo
Flux capacitor	
Transporter	
sc2_boolean	
sc2_scan	No scan [✓ Linear] Random Explicit; Min: 1.00 Max: 100.0 #Points: 10
sc2_enum	4

Minimum level: INFO   freetext filter...

```
Level   Source     Time           Message
INFO    worker(1)  11/09 20:03:45 print:ping 188
INFO    worker(1)  11/09 20:03:46 print:ping 189
INFO    worker(1)  11/09 20:03:47 print:ping 190
ERROR   worker(11) 11/09 20:03:47 root:logging test: error
WARNING worker(11) 11/09 20:03:47 root:logging test: warning
INFO    worker(11) 11/09 20:03:47 root:logging test: info
INFO    worker(11) 11/09 20:03:47 print:None
INFO    worker(11) 11/09 20:03:47 print:True
INFO    worker(11) 11/09 20:03:47 print:foo
INFO    worker(11) 11/09 20:03:47 print:0.003
```

RID	Pipeline	Status	Prio	Due date	Revision	File	Class
12	main	pending	2	11/09 20:04:35	555f317d9fcd970dcd19202cccf60c837eaf9085 initial commit	flopping_f_simulation.py	Floppi
1	main	running	0		3424c9393f26f06bb80b73d9177c71ef20be8a8e fixing all the bugs	run_forever.py	RunFo
2	main	prepare_done	0		3424c9393f26f06bb80b73d9177c71ef20be8a8e fixing all the bugs	run_forever.py	RunFo

Submitted RID 11

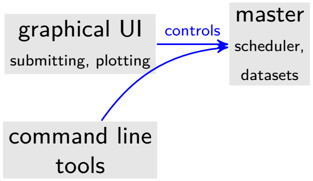
ARTIQ components

master

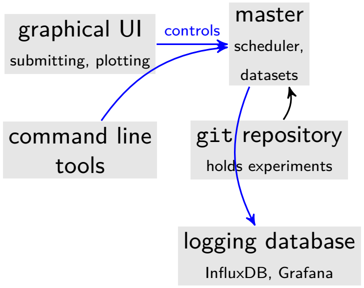
scheduler,

datasets

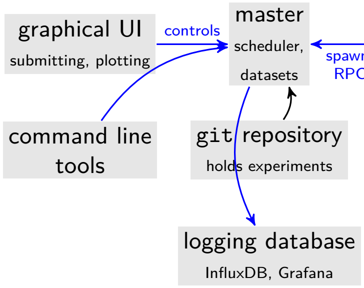
ARTIQ components



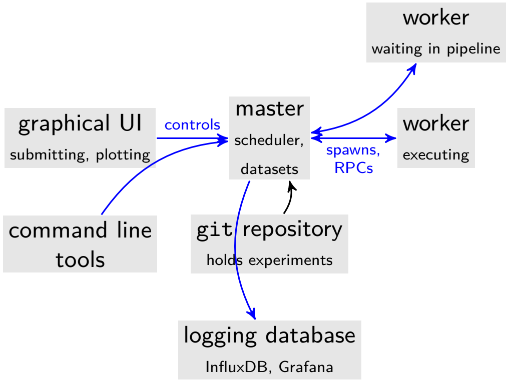
ARTIQ components



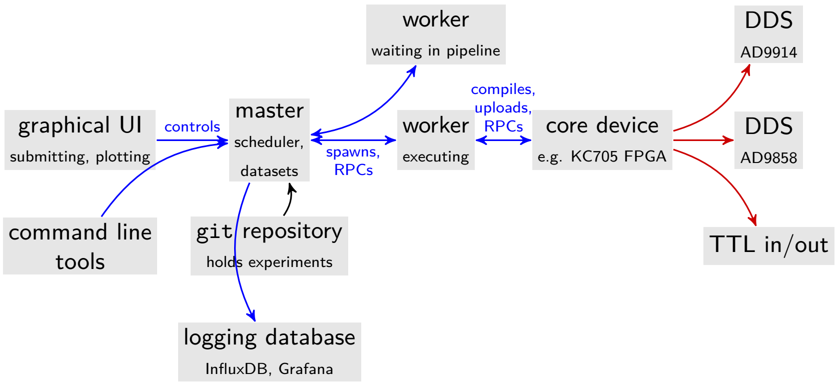
ARTIQ components



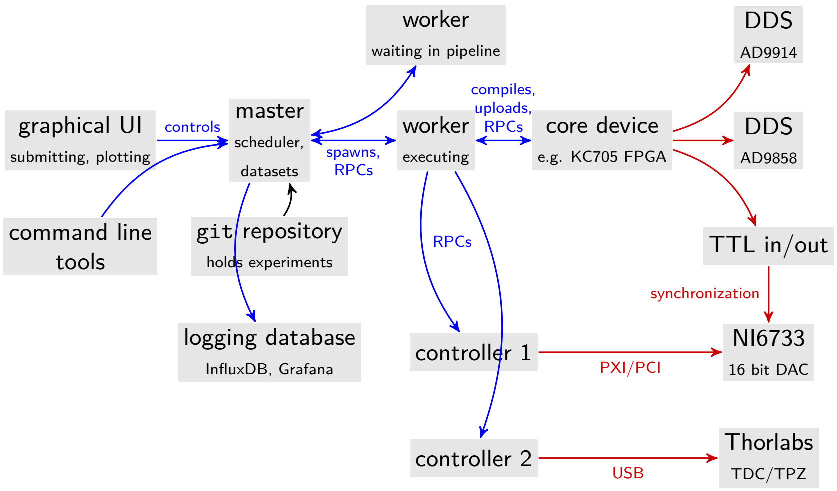
ARTIQ components



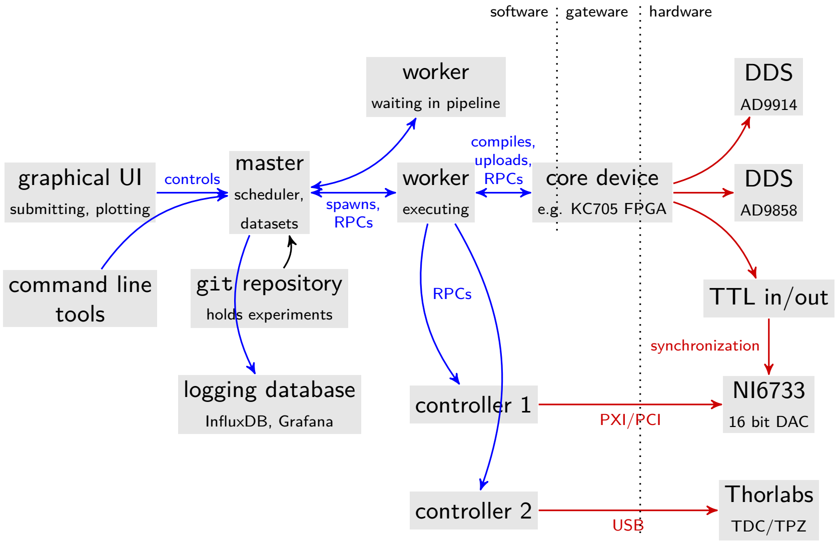
ARTIQ components



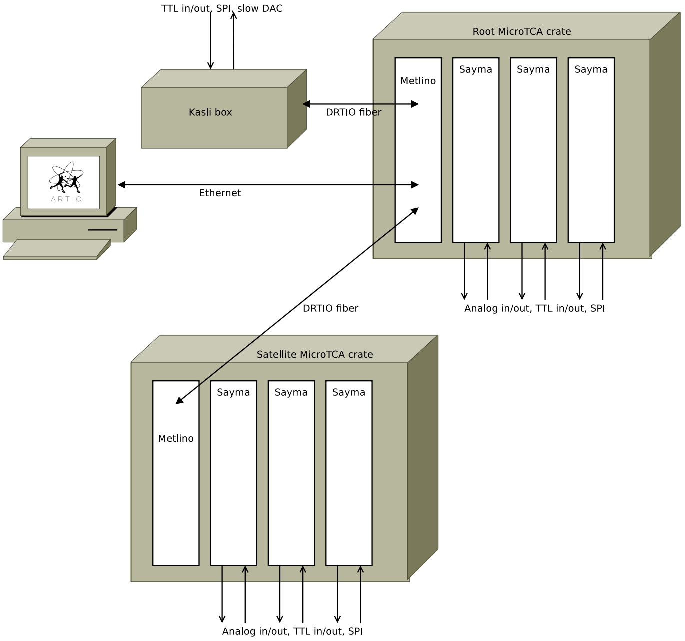
ARTIQ components



ARTIQ components



SINARA



TTL in/out, SPI, slow DAC
Kasli box
DRTIO fiber
Root MicroTCA crate
Metlino
Sayma
Sayma
Sayma
ARTIQ
Ethernet
Analog in/out, TTL in/out, SPI
DRTIO fiber
Satellite MicroTCA crate
Sayma
Sayma
Sayma
Metlino
Analog in/out, TTL in/out, SPI

Define a simple timing language

```
trigger.sync()                # wait for trigger input
start = now()                 # capture trigger time
for i in range(3):
    delay(5*us)
    dds.pulse(900*MHz, 7*us)  # first pulse 5 µs after trigger
at(start + 1*ms)              # re-reference time-line
dds.pulse(200*MHz, 11*us)     # exactly 1 ms after trigger
```

- Written in a subset of Python
- Executed on a CPU embedded on a FPGA (the *core device*)
- `now()`, `at()`, `delay()` describe time-line of an experiment
- Exact time is kept in an internal variable
- That variable only loosely tracks the execution time of CPU instructions
- The value of that variable is exchanged with the RTIO fabric that does precise timing

Convenient syntax additions

```
with sequential:
    with parallel:
        a.pulse(100*MHz, 10*us)
        b.pulse(200*MHz, 20*us)
    with parallel:
        c.pulse(300*MHz, 30*us)
        d.pulse(400*MHz, 20*us)
```

- Experiments are inherently parallel: simultaneous laser pulses, parallel cooling of ions in different trap zones
- `parallel` and `sequential` contexts with arbitrary nesting
- `a` and `b` pulses both start at the same time
- `c` and `d` pulses both start when `a` and `b` are both done (after 20 µs)
- Implemented by inlining, loop-unrolling, and interleaving

Physical quantities, hardware granularity

```
n = 1000
dt = 1.2345*ns
f = 345*MHz

dds.on(f, phase=0)                   # must round to integer tuning word
for i in range(n):
    delay(dt)                        # must round to native cycles

dt_raw = time_to_cycles(dt)          # integer number of cycles
f_raw = dds.frequency_to_ftw(f)      # integer frequency tuning word

# determine correct phase despite accumulation of rounding errors
phi = n*cycles_to_time(dt_raw)*dds.ftw_to_frequency(f_raw)
```

- Need well defined conversion and rounding of physical quantities (time, frequency, phase, etc.) to hardware granularity and back
- Complicated because of calibration, offsets, cable delays, non-linearities
- No generic way to do it automatically and correctly
- $\rightarrow$ need to do it explicitly where it matters

Invite organizing experiment components and code reuse

```
class Experiment:
    def build(self):
        self.ion1 = Ion(...)
        self.ion2 = Ion(...)
        self.transporter = Transporter(...)

    @kernel
    def run(self):
        with parallel:
          self.ion1.cool(duration=10*us)
          self.ion2.cool(frequency=...)
        self.transporter.move(speed=...)
        delay(100*ms)
        self.ion1.detect(duration=...)
```

RPC to handle distributed non-RT hardware

```
class Experiment:
    def prepare(self):                  # runs on the host
        self.motor.move_to(20*mm)   # slow RS232 motor controller

    @kernel
    def run(self):                      # runs on the RT core device
        self.prepare()              # converted into an RPC
```

- When a kernel function calls a non-kernel function, it generates a RPC
- The callee is executed on the host
- Mechanism to report results and control slow devices
- The kernel must have a loose real-time constraint (a long `delay`) or means of re-synchronization to cover communication, host, and device delays

Kernel deployment to the core device

- RPC and exception mappings are generated
- Constants and small kernels are inlined
- Small loops are unrolled
- Statements in parallel blocks are interleaved
- Time is converted to RTIO clock cycles
- The Python AST is converted to LLVM IR
- The LLVM IR is compiled to OpenRISC machine code
- The OpenRISC binary is sent to the core device
- The runtime in the core device links and runs the kernel
- The kernel calls the runtime for communication (RPC) and interfacing with core device peripherals (RTIO, DDS)

ARTIQ compiler

```
parsetree = parse_buffer(source_buffer)
typedtree = asttyped_rewriter.visit(parsetree)
inferencer.visit(typedtree)
cast_monomorphizer.visit(typedtree)
int_monomorphizer.visit(typedtree)
inferencer.visit(typedtree)
monomorphism_validator.visit(typedtree)
escape_validator.visit(typedtree)
iodelay_estimator.visit_fixpoint(typedtree)
constness.visit(typedtree)
devirtualization.visit(typedtree)
artiq_ir = artiq_ir_generator.visit(typedtree)
artiq_ir_generator.annotate_calls(devirtualization)
dead_code_eliminator.process(artiq_ir)
interleaver.process(artiq_ir)
local_access_validator.process(artiq_ir)
invariant_detection.process(artiq_ir)
return llvm_ir_generator.process(artiq_ir)
```

ARITQ compiler

```python
x = 0
for i in range(2*1000*1000*1000):
    x = (x + i) % 101
print(x)
```

```
$ clang -O2 sum.c -o sum
$ time ./sum
real    0m13.725s

$ cat sum.py | python -m artiq.compiler.testbench.jit
time 15.934627326998452
```

pythonparser

- Python source parser/rewriter/emitter/diagnostic engine
- Ideal for tooling

pythonparser

- Python source parser/rewriter/emitter/diagnostic engine
- Ideal for tooling
- Has its own AST, similar to Python's builtin ast

pythonparser

- Python source parser/rewriter/emitter/diagnostic engine
- Ideal for tooling
- Has its own AST, similar to Python's builtin ast
- All versions python 2.6–3.5

pythonparser

- Python source parser/rewriter/emitter/diagnostic engine
- Ideal for tooling
- Has its own AST, similar to Python's builtin ast
- All versions python 2.6–3.5
- Cross-parsing: parse python 2.7 code with a python 3.5 runtime (and rewrite it)

pythonparser

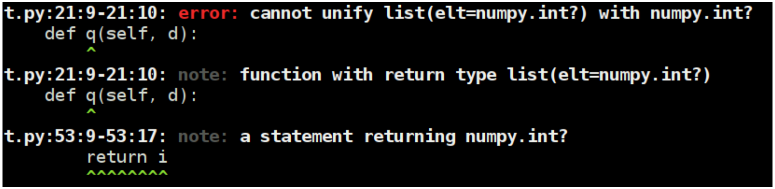
- Python source parser/rewriter/emitter/diagnostic engine
- Ideal for tooling
- Has its own AST, similar to Python's builtin ast
- All versions python 2.6–3.5
- Cross-parsing: parse python 2.7 code with a python 3.5 runtime (and rewrite it)
- Pure python: build self-hosting tools with it!
- Rewrite engine for altering code (refactoring tools)

pythonparser

Precise location/range information for every token, including commets, sub-nodes

pythonparser

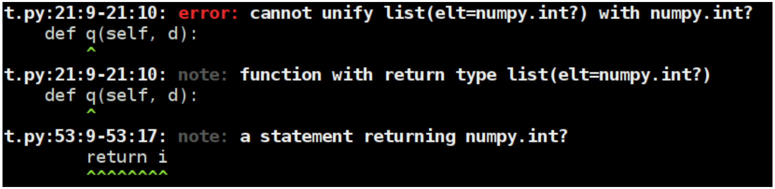
Precise location/range information for every token, including commets, sub-nodes



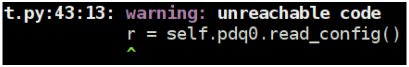
```
t.py:21:9-21:10: error: cannot unify list(elt=numpy.int?) with numpy.int?
    def q(self, d):
        ^
t.py:21:9-21:10: note: function with return type list(elt=numpy.int?)
    def q(self, d):
        ^
t.py:53:9-53:17: note: a statement returning numpy.int?
        return i
        ^^^^^^^^
```

pythonparser

Precise location/range information for every token, including commets, sub-nodes



```
t.py:21:9-21:10: error: cannot unify list(elt=numpy.int?) with numpy.int?
    def q(self, d):
        ^
t.py:21:9-21:10: note: function with return type list(elt=numpy.int?)
    def q(self, d):
        ^
t.py:53:9-53:17: note: a statement returning numpy.int?
        return i
        ^^^^^^^^
```



```
t.py:43:13: warning: unreachable code
            r = self.pdq0.read_config()
            ^
```

pythonparser

- `https://github.com/m-labs/pythonparser`, MIT license
- grumpy: Python to Go source code transcompiler and runtime

pythonparser

- `https://github.com/m-labs/pythonparser`, MIT license
- grumpy: Python to Go source code transcompiler and runtime
  - Developed by Google to scale youtube Python frontend (CPython2.7)
  - Transpile Python and use the Go runtime
  - Avoid Python concurrency issues

```
$ build/bin/grumpc hello.py > hello.go
$ go build -o hello hello.go
```

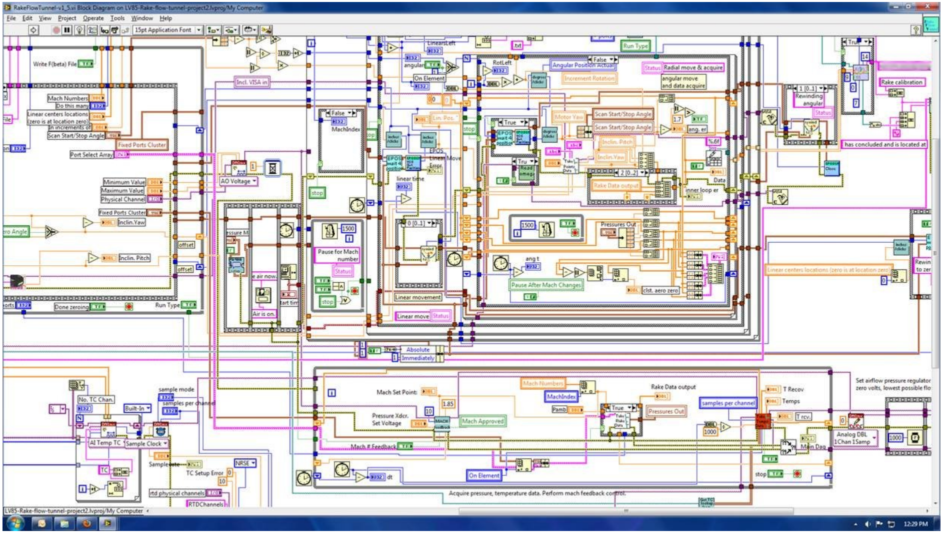
pythonparser

- `https://github.com/m-labs/pythonparser`, MIT license
- grumpy: Python to Go source code transcompiler and runtime
  - Developed by Google to scale youtube Python frontend (CPython2.7)
  - Transpile Python and use the Go runtime
  - Avoid Python concurrency issues

```
$ build/bin/grumpc hello.py > hello.go
$ go build -o hello hello.go
```

- ARTIQ

(Experimental) physicists and computers...



They call it a "high-viscosity language"