Making CPython Fast With Trace-Based Optimisations

Mark Shannon

Who Am I?

- Mark Shannon
- Recently completed PhD @ Glasgow University
  - “Construction of High-Performance Virtual Machines for Dynamic Languages”
  - HotPy = **H**otPy **O**ptimising **T**racing **Py**thon
- Interested in all aspects of Virtual Machines
- Especially VMs for Dynamic Languages

This Talk

- CPython – What it is, why it is important
- Tracing – How and why it works
- Problems of tracing in Python
- Solving these problems, by dropping to a lower level
- Optimising Traces
  - Specialisation and Deferred Object Creation
- Other Issues (depending on time available)
- Conclusions

CPython VM

- Default Python implementation
- Over 20 years old; has evolved with the Python language
- Based on 1980s technology
- I want to bring CPython into the 21st century
- **No** change to language
- **No** change to C API

Why CPython?
(Why not just use PyPy?)

- CPython has many C extensions
  - Numpy, scipy
- CPython can be embedded in other applications
  - Blender
- Runs on many platforms

What is Tracing?

- Several meanings, but in this context it means:
- Monitor the execution of the program until a “hot” point is found
- Record the execution of the hot traces in the program (Traces may cross function calls)
- Optimise the recorded traces and save in a trace cache
- When the same part of the program is next executed, the optimised trace is executed instead of the original bytecode.

Modifying CPython for Tracing

- Separate VM from hardware-machine
  - CPython makes a C call for each Python call
  - Calls to Python code should stay within the interpreter, not call into C code
- Break calls into prepare/call pair:
  - Prepare = create new frame, fill in parameters, etc.
  - Call = save return address, and jump to start of function
- Prepare part can call out to C code
- Call part stays within interpreter

Managing Traces

- Maintain a cache of traces.
- Each trace has a hotness
- trace.hotness += 1; when a trace is executed
- trace.hotness *= 0.75 every *T*ms.
  - Set *T* experimentally (probably 200 to 500ms)
- Two thresholds: warm & cold.
- If hotness > warm then add new trace to cache
- If hotness < cold then evict trace from cache

Optimising Traces

- Want to use bytecode both as input and output
- Allows optimisations to be tested independently
- Can develop optimisers incrementally.
- Important optimisations:
  - Specialisation
  - Escape analysis/Deferred object allocation
  - Compilation (Removal of interpretive overhead)

Problems with Tracing Bytecode

- Too high level
  - Each bytecode does too much, hard to optimise
- Not atomic
  - Can observe VM state during the execution of the bytecode
  - VM state includes internal state of interpreter
  - Impossible to record trace

Lower-Level Bytecodes

- Will need to add new bytecodes.
- These bytecodes are only used internally.
- Specialised forms of bytecodes:
  - Integer & floating-point arithmetic
  - Bytecodes for directly calling C functions
- Building-blocks for Python semantics:
  - Creating and initialising frames
  - Finding attributes in class and object dicts

Sub-Python Abstract Machine

SPAM

- Lower level instructions than CPython
- All CPython bytecodes can be implemented in SPAM instructions.
- Higher level than the hardware, includes instructions for core operations in Python:
  - Load/store in object dictionary
  - Load from class (and super-classes') dictionary
  - Python-aware call primitive
- All SPAM instructions are *atomic*.

SPAM instructions

- SPAM includes low-level instructions:
  - iadd, fadd
  - native_call
- And special instructions to support Python semantics:
  - load_special_or_goto
  - class_of
  - from_dict_or_goto
  - swap_exception_state

Tracing with SPAM

- Record CPython bytecode for common cases
- Otherwise drop to SPAM level.
- Example BINARY_ADD
  - Record BINARY_ADD for int, float, etc.
  - Trace call to SPAM code for add otherwise.
- SPAM equivalents for *all* CPython bytecodes
  - One-to-one equivalence for simple bytecodes

SPAM equivalents

```
LOAD_ATTR =
  load_special_or_goto '__getattribute__', impossible
  name
  CALL_FUNCTION 1  0
```

- Dispatches to object.__getattribute__
- CPython does this in C:
  - C is opaque and cannot be optimised
  - SPAM code is transparent and optimisable.

SPAM functions (1)

```
BINARY_SUBSCR =
    LOAD_CONST binary_subscr_function
    ROT_THREE
    CALL_FUNCTION 2  0
```

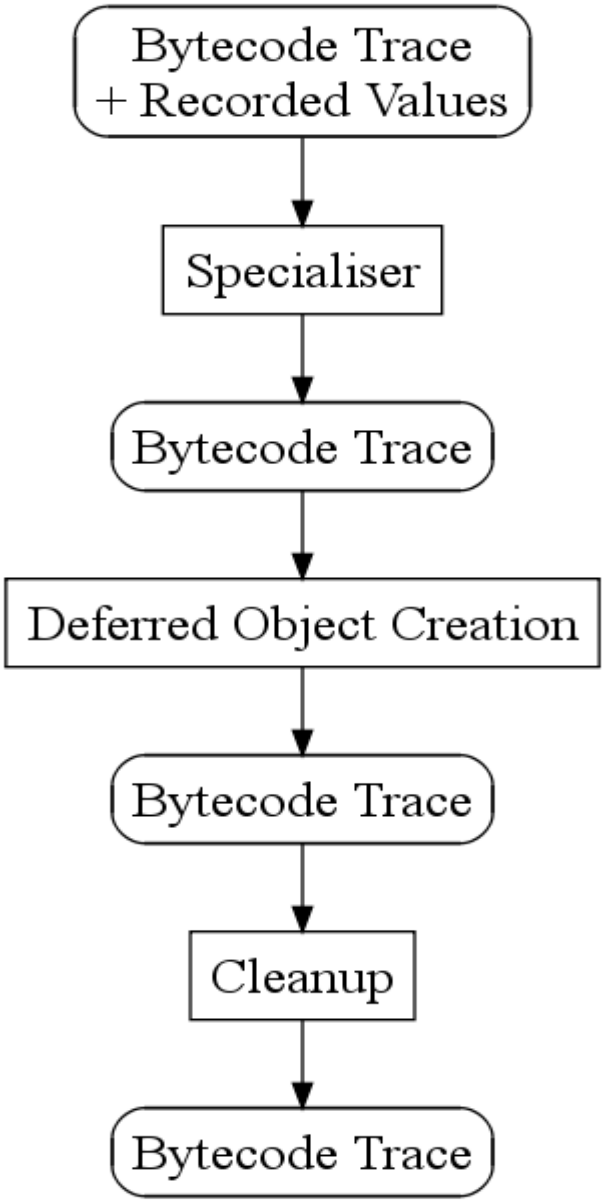
SPAM functions (2)

```
def binary_subscr_function:
  load_frame 0
  load_special_or_goto '__getitem__', error
  load_frame 1
  CALL_FUNCTION 1  0
  RETURN_VALUE
error:
  load_constant not_subscriptable_error
  load_frame 0
  CALL_FUNCTION 1  0
  RETURN_VALUE
```

The Main Trace-Based Optimisations

- Specialisation
  - Customise code for the observed types of variables.
- Deferred Object Creation
  - Don't create objects until the last possible moment, can avoid creating a lot of objects altogether
- Compilation
  - Remove interpretative overhead
  - Ouputs machine code

Optimiser Chain



Specialisation

- Customise the trace for expected types.
- For each bytecode:
  - Ensure type of objects is as expected
  - Replace bytecode with faster equivalent
  - Update type information
- Use *guards* to ensure types are expected
- Can replace slow bytecodes with fast ones
  - BINARY_ADD → int_add

Guards

- Inline Guards
  - Extra bytecodes inserted into the trace
- Out-of-Line Guards
  - Extra code is added elsewhere to invalidate trace if asumptions are violated.
  - Example: To guard against a class attribute changing, add code to type.__setattr__

Deferred Object Creation

- Defer the creation of objects for as long as possible
- Can often defer creation for ever, as the objects are never needed
- Common for objects created to pass parameters and in loops

Deferred Object Creation (2)

- Maintain shadow stacks
- 3 independent shadow stacks:
  - Shadow data stack
  - Shadow frame stack
  - Shadow exception-handling stack
- Only when an object on a stack is actually needed are the instructions to create it emitted

Specialisation and D.O.C. Example

- Calling a simple function, that returns its only parameter
- E.g. The __iter__ method of any iterator

```
def __iter__(self):
    return self

it = x.__iter__()
```

Starting Trace

- LOAD_FAST 2 (x)
- load_special '__iter__'
- BUILD_PARAMETERS 0 0
- prepare_bm_call # bm, t, d -> f, (self,)+t, d
- func_check 2 id(x.__iter__)
- MAKE_FRAME
- INIT_FRAME
- LOAD_FAST 0 (self)
- RETURN_VALUE

Specialisation (1)

- LOAD_FAST 2 (x)
- load_special '__iter__'
  - Ensure type of x (A guard may need to be inserted)
  - Add out-of-line guards to protect against redefinition of __iter__
  - Replace with 'bind' instruction to create bound-method
- BUILD_PARAMETERS 0 0
- prepare_bm_call

Specialisation (2)

- func_check 2 id(X.__iter__)
  - Due to guards inserted earlier, this can be removed
- MAKE_FRAME
- INIT_FRAME
- LOAD_FAST 0 (self)
  - Guaranteed to be defined, replace with 'load_frame'
- POP_FRAME

Specialiser Output

- LOAD_FAST 2 (x)
- ensure_type id(X)
- bind id(X.__iter__)
- BUILD_PARAMETERS 0 0
- prepare_bm_call
- MAKE_FRAME
- INIT_FRAME
- LOAD_FAST 0 (self)
- POP_FRAME

Specialiser

- Makes traces faster
- Small reduction in number of bytecodes
- Specialised bytecodes are faster than the ones they replaced.

Deferred Object Creation (1)

- LOAD_FAST 2 (x)
  - Defer load. Push local[2] (x) to shadow data stack
- bind id(X.__iter__)
  - Pop local[2] from shadow data stack
  - Push bound-method (local[2], X.__iter__) to shadow data stack
- BUILD_PARAMETERS 0 0
  - Push empty-tuple () and empty-dict {} to shadow stack

D.O.C. (2)

- prepare_bm_call
  - Pop all three values from shadow data stack and rearrange: bm(local[2], X.__iter__), (), {} => X.__iter, (local[2],), {}
- MAKE_FRAME
  - Examine callable on data stack (X.__iter__)
  - Push a new frame to the shadow frame stack.

D.O.C. (3)

- INIT_FRAME
  - Pop all 3 values from the data stack
  - Initialise shadow frame from values:
  - shadow_frame(deferred_local[0] = local[2])
- LOAD_FAST 0 (self)
  - Push local[2] (deferred_local[0] == local[2])
- POP_FRAME
  - Discard deferred frame

D.O.C. (4)

- So far have emitted zero instructions.
- If trace ends, must materialise the stack:
  - Emit on bytecode: LOAD_FAST 2
- Converted nine bytecodes to one!
- Contrived example, but D.O.C. can often reduce the size of code by 50% or more.

Cleanup

- D.O.C is very aggressive, and can introduce quite a lot of stores and loads.
- Tidy up afterwards
- Remove store/load pairs, etc.

Compilation

- Final (optional) stage
- Add third level of “hotness”, *hot*.
- When trace becomes hot, then compile it.
- Translate traces to machine-code.
- Previous passes have lowered the level of the traces; mainly loads/stores and primitive ops.
- Translation to machine code is straight-forward
- Use LLVM, libJIT, nanoJIT, roll-your-own...

How Much Faster?

- It depends:
  - If your code spends 90% of its time doing I/O it won't make any difference.
- Estimates based on my HotPy VM
- For computational Python, *very* roughly:
  - Interpreter only x3
  - With compilation x10
- For Web stuff – I don't know

Other Issues

- Garbage Collection
- Reimplementing the dict for optimisation
- Enhancing the builtin function type
- Object Representation (Tagging of values)

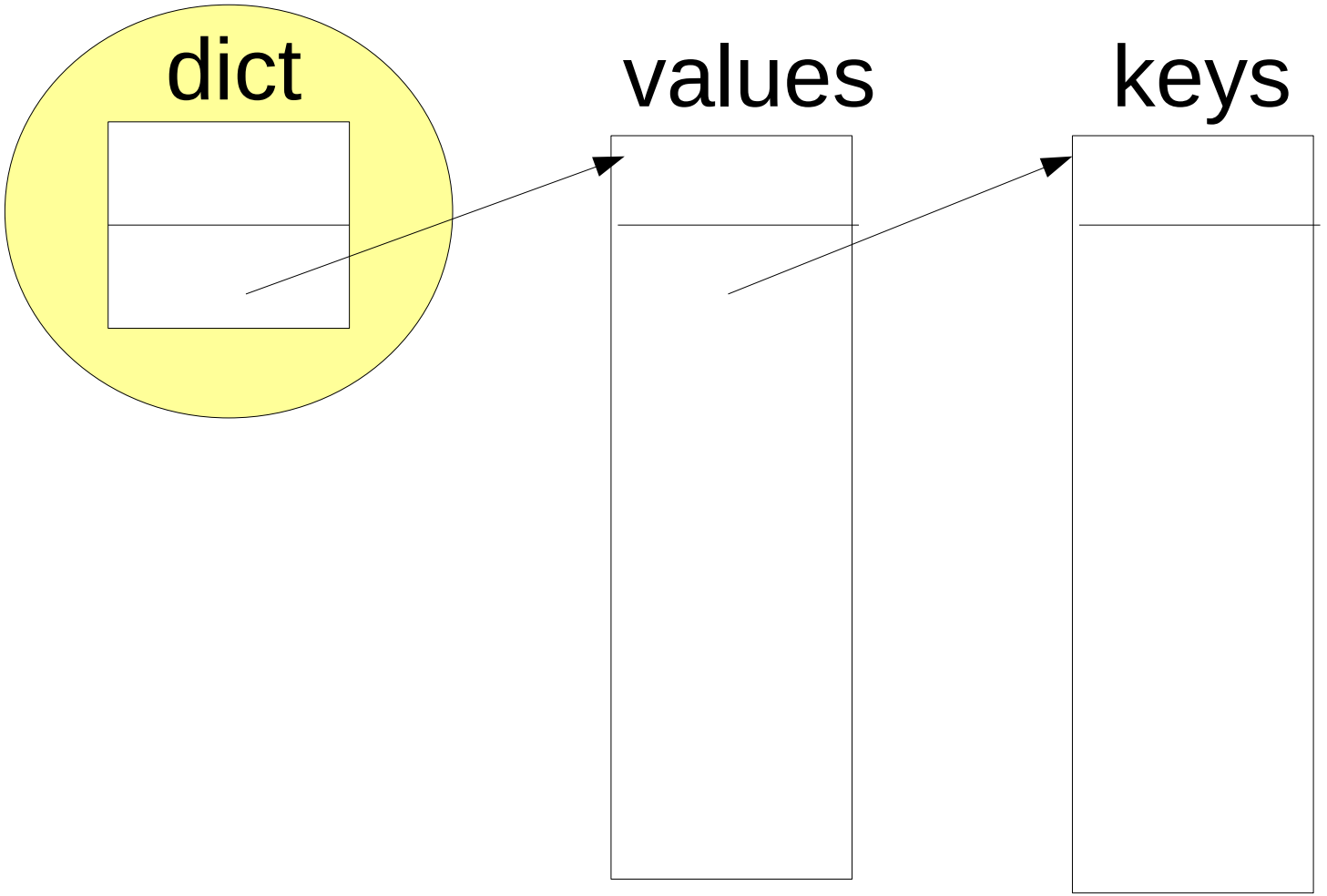
Garbage Collection

- Reference counting **is** garbage collection
- CPython can keep ref-counting for extensions
- Use “tracing” GC for internal objects
  - Much faster allocation
  - Less overhead
  - May induce slightly longer pauses
- **Or** use “tracing” for stack and locals only
  - Gain some performance benefit
  - Smaller changes required

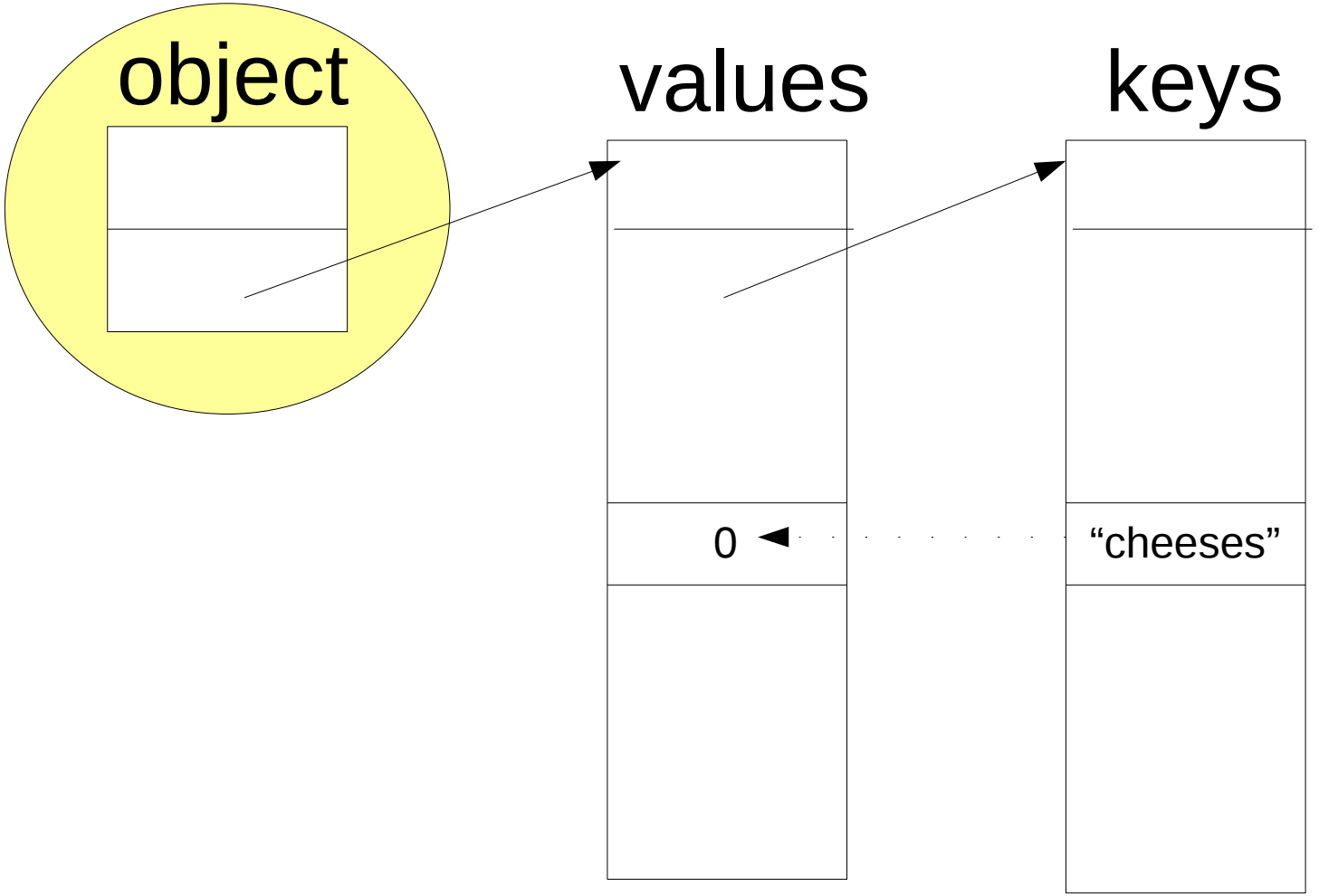
Reimplementing **dict**

- **dict** is heavily optimised for general case
- Three categories of **dict** usage:
  - Global/module variable lookup
  - Object attribute lookup
  - Explicit dictionary use
- Specialisation removes **dict** lookup for globals
- Reduce memory use by sharing keys
- Fast as slot access to object attributes

Reimplementing **dict** (2)



Reimplementing **dict** (3)



- ensure(x.values.keys == keys)
- load_slot(x.values, offsetof(cheeses))

Enhancing builtin function

- Currently (at least) four different types
- Refactor into two types
  - functions and unbound methods
- Generalise the allowed parameter formats
- Add parameter type information
  - VM should guarantee that parameters are correct
  - Faster (optimisation can eliminate checks)
  - Simpler, checks in one place only (DRY principle)

Object Representation

- Reduce header size
  - Saturating ref-count, type as integer ID.
- Tagging
  - Take a few bits out of word for type information
  - Fits almost all ints into pointer (no need for boxing)
  - Can even do the same for floats on 64 bit machine
- Embed type-id in pointer (this is a bit extreme!)
- Evaluation required – May get slower or faster

Conclusion

- CPython can be modified to support tracing
- Tracing allows powerful optimisations
  - Specialisation
  - Deferred Object Creation
- Can be made faster and remain portable
  - Optimised Traces can be interpreted quickly
- Compilation will make it even faster

Thank You For Listening

- Any Questions?
- For more info about the experimental platform and the optimisations:
- www.hotpy.org
- Or search for “HotPy”