



HotPy, A Comparison



University
of Glasgow

HotPy, PyPy And Unladen-Swallow

An overview of high-performance
virtual machines for Python

Mark Shannon

University of Glasgow



HotPy, A Comparison



University
of Glasgow

- Introduction to the VMs
- Think in terms of time rather than speed
- Execution overheads
- How the different VMs remove these overheads
- Other considerations – Concurrency, API.
- Conclusion



The VMs

- HotPy
 - Built with the GVM T as part of my research
- PyPy
 - “psyco done right”
 - PyPy-VM built using PyPy “translation chain”
- Unladen Swallow
 - Branch of CPython
 - Uses LLVM for JIT compilation



HotPy

- Built with Glasgow Virtual Machine Toolkit
- GVMT manages correct interaction between interpreter, compiler and GC.
- Builds interpreter and JIT compiler from common source.
- GVMT built compiler uses LLVM for machine-code generation



PyPy

- PyPy is both a VM and a tool for building VMs.
- PyPy toolchain translates PyPy-VM (written in RPython) to C (or JVM/CLI) code.
- JIT compiler produced by toolchain.



Unladen Swallow

- Manually created JIT compiler
- Converts CPython bytecode to LLVM bitcode
- LLVM optimises and generates machine-code



Speed = 1 / Time



- Thinking in terms of speed can be misleading
- Why? Poor estimation of performance
 - “JIT compilers give x10 speedup for Java”
 - “Should be able to get half that for Python”
 - “Expect to get a x5 speedup?”



$$\text{Time} = 1 / \text{Speed}$$



- Better to think in terms of time
- Focuses on removing overhead
- Thinking in term of time:
 - x10 speedup = -90% execution time
 - Half of that is -45% execution time
 - Results in x1.8 speedup



Execution Overheads



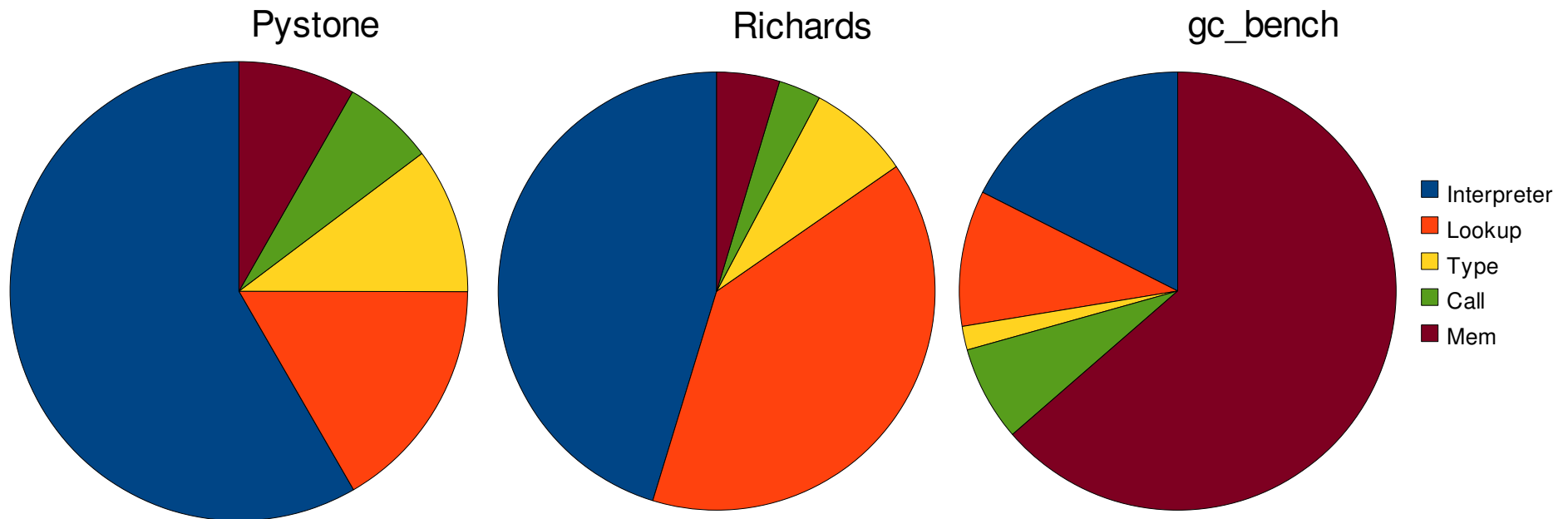
University
of Glasgow

- Interpretive
- Imprecise type information
- Parameter Handling & Call overhead
- Lookups (globals/builtins/attributes)
- Memory management (garbage collection)
- (This is not a definitive list)



Example Overheads

- Profiling CPython for three benchmarks





Interpretive Overhead



University
of Glasgow

- The reason java interpreters are slower than java compilers
- Caused by:
 - Processor stalls
 - Redundant moves to and from the stack
 - Short code sequences prevent optimisation



Imprecise Type Information



University
of Glasgow

- Tagging or boxing is required
- Operator and method lookup required



Calls and Parameters



University
of Glasgow

- Packing of parameters into tuple and dict
- Checking parameter format at call
- Unpacking of parameters
- Frame allocation/deallocation
 - Frames are big objects



Lookups

- Globals, builtins and attributes
- Repeated lookups to find location in memory
- Treating constants as variables prevents other optimisations



Memory Management



University
of Glasgow

- Object allocation
 - Allocating space for new objects
- Garbage collection
 - Finding live objects and recycling dead ones
- Main overhead is not allocation/collection but *inefficient* allocation/collection



Constraints on Optimisation(1)



University
of Glasgow

- Must keep the same semantics



Constraints on Optimisation(2)



University
of Glasgow

- Must keep the same semantics
- Concurrency



Constraints on Optimisation(3)



University
of Glasgow

- Must keep the same semantics
- Concurrency
- API and 3rd party code



Constraints on Optimisation(4)



University
of Glasgow

- Must keep the same semantics
- Concurrency
- API and 3rd party code
- Concurrency



Some Terminology

- Tracing
 - Recording the flow of the program “as it happens”
- Specialising
 - Converting a general form of a program to a form optimised for a particular set of values
 - Eg. `add` → `int_add`, for integer values



HotPy

- HotPy
 - Tracing, specialising **interpreter**
 - Trace and optimise “warm” code (~20)
 - Interpret optimised bytecode
 - JIT compiles “hot” code (~1000)
 - Potential to compile while optimised code is being interpreted



PyPy



University
of Glasgow

- PyPy
 - Tracing and specialising JIT **compiler**
 - Tracing occurs “below” level of interpreter
 - Effectively combines tracing and specialisation into one.
 - Sophisticated techniques to recover from trace-exits mid bytecode



Unladen Swallow

- Whole function optimisation.
 - Like the Sun HotSpot™ JVM
 - Does Python specific specialisations during conversion to LLVM code
 - Enhances LLVM optimisers to better understand CPython VM



Guards

- Many optimisations are speculative
- Need to guard code against incorrect guesses
- Guards can be inline
 - Guard is placed directly before code
- Or out-of-line
 - Guard is placed to intercept changes to certain values



Execution Overheads (Reprise)



University
of Glasgow

- Interpretive
- Imprecise type information
- Parameter Handling & Call overhead
- Other Lookups (globals/builtins)
- Memory management (garbage collection)
- (This is not a definitive list)



(JIT) Compilation



University
of Glasgow

- Removes interpretive overhead
- Converts bytecode to machine-code
- Removes interpretive overhead
- HotPy, PyPy and Unladen-Swallow all compile



Better type precision



University
of Glasgow

- HotPy gathers type-information as it traces
- PyPy traces at a lower-level, so type-information is embedded in the trace
- Unladen-Swallow samples types before compilation.



Call handling



- HotPy
 - Trace include both the caller and callee.
 - Packing, unpacking and parameter checks can be removed
- PyPy
 - As for HotPy
- Unladen Swallow
 - Cannot do Python-specific optimisations
 - Uses LLVM to inline some functions



Lookups(1)



University
of Glasgow

- Global and builtin variables
- Class attributes
- All three optimise these
 - All three use inline guards
 - HotPy and Unladen Swallow also use out-of-line guards



Lookups(2)

- Instance attributes (in object dict)
- HotPy
 - Uses techniques from prototype languages
 - All objects of the same class share dict keys
- PyPy
 - Different technique, but with a similar effect
- Unladen Swallow
 - No optimisation



Memory Management



University
of Glasgow

- HotPy and PyPy
 - Generational GC
 - Fast allocation
 - Zero time collection of short-lived objects
 - Fast collection of long-lived objects
- Unladen Swallow
 - Uses CPython's reference-counting, with the optimiser removing some inc/dec pairs



Memory Management



University
of Glasgow

- HotPy GC is *much* faster than CPython
 - x10 for long-lived objects
 - x100 for short-lived objects
- PyPy's GC is about as fast HotPy's GC



API & 3rd Party Code



University
of Glasgow

- HotPy
 - Future work
 - GC supports pinning so access to objects can be direct
 - Reference count embedded in object
 - Moderate cost to call from VM to 3rd party code



API & 3rd Party Code



University
of Glasgow

- PyPy
 - Ongoing development
 - GC is a moving collector, access to objects through handle
 - Reference count kept separately
 - Higher cost to call from VM to 3rd part code
 - Future optimisations may reduce overhead



API & 3rd Party Code



University
of Glasgow

- Unladen Swallow
 - Fully implemented. 100% conformance
 - Reference counting used internally
 - Very low cost to call from VM to 3rd party code



Concurrency(1)

- HotPy
 - No GIL. Truly concurrent
 - dict designed to support limited concurrent access
 - Fast locks based on locks developed for the JVM
 - GIL still required for 3rd party code, but VM threads can run concurrently



Concurrency(2)

- PyPy
 - Has GIL. Very limited concurrency
 - GIL could be removed
 - Care needed to ensure optimisations remain valid
- Unladen Swallow
 - Has GIL. Very limited concurrency
 - Reference-counting prevents removal of GIL



How Fast?

- It depends.
- Remember “Speed = 1 / Time”
 - If only 50% of execution time can be optimised:
 - Then speedup must be less than x2
- Pure Python code x15 to x20 speedup?
- But no Python program is pure Python
- Concurrency can yield further speed ups...



Questions?



University
of Glasgow

- <http://code.google.com/p/hotpy>
- <http://code.google.com/p/gvmt>
- <http://www.dcs.gla.ac.uk/~marks>