University of Glasgow

Shannon, Mark (2011) *The construction of high-performance virtual machines for dynamic languages.* PhD thesis.

http://theses.gla.ac.uk/2975/

Copyright and moral rights for this thesis are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Glasgow Theses Service
http://theses.gla.ac.uk/
theses@gla.ac.uk

University of Glasgow

The Construction of High-Performance Virtual Machines for Dynamic Languages

Mark Shannon, MSc.

Submitted for the Degree of
Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

November 2011

Abstract

Dynamic languages, such as Python and Ruby, have become more widely used over the past decade. Despite this, the standard virtual machines for these languages have disappointing performance. These virtual machines are slow, not because methods for achieving better performance are unknown, but because their implementation is hard. What makes the implementation of high-performance virtual machines difficult is not that they are large pieces of software, but that there are fundamental and complex interdependencies between their components. In order to work together correctly, the interpreter, just-in-time compiler, garbage collector and library must all conform to the same precise low-level protocols.

In this dissertation I describe a method for constructing virtual machines for dynamic languages, and explain how to design a virtual machine toolkit by building it around an abstract machine. The design and implementation of such a toolkit, the Glasgow Virtual Machine Toolkit, is described. The Glasgow Virtual Machine Toolkit automatically generates a just-in-time compiler, integrates precise garbage collection into the virtual machine, and automatically manages the complex interdependencies between all the virtual machine components.

Two different virtual machines have been constructed using the GVMT. One is a minimal implementation of Scheme; which was implemented in under three weeks to demonstrate that toolkits like the GVMT can enable the easy construction of virtual machines. The second, the HotPy VM for Python, is a high-performance virtual machine; it demonstrates that a virtual machine built with a toolkit can be fast and that the use of a toolkit does not overly constrain the high-level design. Evaluation shows that HotPy outperforms the standard Python interpreter, CPython, by a large margin, and has performance on a par with PyPy, the fastest Python VM currently available.

Contents

1 Introduction **11**
1.1 Virtual Machines . . . 12
1.2 Dynamic Languages . . . 12
1.3 The Problem . . . 13
1.4 Thesis . . . 14
1.5 Contributions . . . 14
1.6 Outline . . . 15

2 Virtual Machines **17**
2.1 A Little History . . . 17
2.2 Interpreters . . . 19
2.3 Garbage Collection . . . 24
2.4 Optimisation for Dynamic Languages . . . 31
2.5 Python Virtual Machines . . . 33
2.6 Other Interpreted Languages and their VMs . . . 36
2.7 Self-Interpreters . . . 43
2.8 Multi-Threading and Dynamic Languages . . . 43
2.9 Conclusion . . . 44

3 Abstract Machine Based Toolkits **45**
3.1 Introduction . . . 45
3.2 The Essential Features of a Virtual Machine . . . 47
3.3 An Abstract Machine for Virtual Machines . . . 48
3.4 Optimisation in VMs for Dynamic Languages . . . 52
3.5 When to use the abstract machine approach? . . . 54
3.6 Alternative Approaches to Building VMs . . . 55
3.7 Related Work . . . 56
3.8 Conclusions . . . 57

4 The Glasgow Virtual Machine Toolkit **59**
4.1 Overview . . . 59
4.2 The Abstract Machine . . . 61
4.3 Front-End Tools . . . 66
4.4 Back-End Tools . . . 71
4.5 Translating GVMT Abstract Machine Code to Real Machine Code 73

4.6 Memory Management in the GVMT 78
4.7 Locks 86
4.8 Concurrency and Garbage Collection 88
4.9 Comparison of PyPy and GVMT 90
4.10 The GVMT Scheme Example Implementation 91
4.11 Conclusions 92

5 HotPy, A New VM for Python 95
5.1 Introduction 95
5.2 The HotPy VM Model 96
5.3 Design of the HotPy VM 99
5.4 Tracing and Traces 104
5.5 Optimisation of Traces 108
5.6 Specialisation 111
5.7 Deferred Object Creation 113
5.8 Further Optimisations 119
5.9 De-Optimisation 120
5.10 An Example 120
5.11 Deviations from the Design of CPython 125
5.12 Dictionaries 126
5.13 Related Work 130
5.14 Conclusion 131

6 Results and Evaluation 133
6.1 Introduction 133
6.2 Utility of the GVMT and Toolkits in General 133
6.3 Performance of the GVMT Scheme VM 134
6.4 Comparison of Unladen Swallow, the PyPy VM, and HotPy . . . 135
6.5 Aspects of Virtual Machine Performance 143
6.6 Memory Usage 146
6.7 Effect of Garbage Collection 149
6.8 Potential for Further Optimisation 150
6.9 Conclusions 152

7 Conclusions 155
7.1 Review of the Thesis 155
7.2 Significant Results 156
7.3 Dissertation Summary 156
7.4 Future Work 158
7.5 In Closing 160

A The GVMT Abstract Machine Instruction Set 162

B The GVMT Abstract Machine Language Grammar 197

C Python Attribute Lookup Semantics 200

C.1 Definitions 200
C.2 Lookup Algorithm 201

D Surrogate Functions **203**
D.1 The __new__ method for tuple 203
D.2 The __call__ method for type 204
D.3 The Binary Operator 204

E The HotPy Virtual Machine Bytecodes **205**
E.1 Base Instructions 205
E.2 Instructions Required for Tracing 213
E.3 Specialised Instructions 216
E.4 D.O.C. Instructions 221
E.5 Super Instructions 222

F Results **224**

Bibliography **228**

List of Tables

2.1 Main Python Implementations 34
2.2 Main Ruby Implementations 39

4.1 GVMT Types 64

6.1 Unoptimised Interpreters. Short Benchmarks 139
6.2 Unoptimised Interpreters. Medium Benchmarks 139
6.3 Full VM. Short Benchmarks 140
6.4 Full VM. Medium Benchmarks 140
6.5 Full VM. Long Benchmarks 141
6.6 Optimised Interpreters. Short Benchmarks 141
6.7 Optimised Interpreters. Medium Benchmarks 141
6.8 Interpreter vs. Compiler. Short Benchmarks 142
6.9 Interpreter vs. Compiler. Medium Benchmarks 142
6.10 Interpreter vs. Compiler. Long Benchmarks 142
6.11 HotPy(C) Performance Permutations. Speeds Relative to CPython 144
6.12 HotPy(Py) Performance Permutations. Speeds Relative to CPython 144
6.13 Speed Up Due to Adding Specialiser; HotPy(C). 144
6.14 Speed Up Due to Adding D.O.C.; HotPy(C). 145
6.15 Speed Up Due to Adding Compiler; HotPy(C). 145
6.16 Speed Up Due to Adding Specialiser; HotPy(Py). 145
6.17 Speed Up Due to Adding D.O.C.; HotPy(Py). 145
6.18 Speed Up Due to Adding Compiler; HotPy(Py). 146
6.19 Memory Usage 147
6.20 CPython GC percentages 149
6.21 Theoretical CPython Speedups 149

F.1 Timings (in seconds); short benchmarks 225
F.2 Timings (in seconds); medium benchmarks 226
F.3 Timings (in seconds); long benchmarks 227

List of Figures

2.1 Token Threaded Code – Stack-based program for A+B 20
2.2 Switch Threaded Code – Stack-based program for A+B 20
2.3 Switch Threaded Implementation in C 21
2.4 Direct Threaded Code – Stack-based program for A+B 21
2.5 Indirect Threaded Code – Stack-based program for A+B 21
2.6 Subroutine Threaded Code – Stack-based program for A+B 22
2.7 Call Threaded Code – Stack-based program for A+B 22
2.8 Memory Cycle 24
2.9 Uncollectable cycle for Reference Counting 27
2.10 Self VM Tag Formats 29

3.1 Generalised Bytecode Optimiser 54
3.2 Toolkit Assisted Optimiser 55

4.1 The GVMT Tools 60
4.2 The GVMT Abstract Machine Model 62
4.3 Tree for `a += b` 67
4.4 The GVMT-built Compiler 72
4.5 The GVMT Compiler Generator 73
4.6 A Memory Zone Consisting of Eight Blocks 79
4.7 Address word (most significant bit to the left) 81
4.8 Lock representations 87
4.9 Lock states 88

5.1 A HotPy Thread 97
5.2 Call sequence 98
5.3 The HotPy Stack 102
5.4 Control of Execution in HotPy 103
5.5 The HotPy Optimiser Chain 109
5.6 Source Code for DOC Example 116
5.7 Trace Without DOC 117
5.8 Trace With DOC 118
5.9 Shadow Stacks For Start of Trace in Figure 5.7 118
5.10 Fibonacci Program 121
5.11 Flowgraph for fib function 122
5.12 Flowgraph for fib_list function 122

5.13 Traces of the Fibonacci Program With an Input of 40 123
5.14 Extended Trace for Overflow 124
5.15 The HotPy dict 128

6.1 Performance of Scheme Implementations 135
6.2 Performance of HotPy and PyPy compared to C and Java 150
6.3 Quality of HotPy and PyPy Optimisations Measured against Java (OpenJDK) 152

List of Algorithms

4.1 Card-marking by object address 83
4.2 Card-marking by slot address 83
4.3 Conventional Card-marking 84
4.4 Naïve Bump-pointer Allocation 85
4.5 Improved Bump-pointer Allocation 85
C.1 Python Attribute Lookup (Objects) 202
C.2 Python Attribute Lookup (Types) 202
C.3 Descriptor Lookup 202

Acknowledgements

First and foremost, I would like to thank Ally Price for her love and support throughout my PhD, and for proof reading, and rereading, several versions of this dissertation.

I would like to thank my supervisor, David Watt, for his diligent and professional supervision.

I would also like to thank the members of the open source community for providing all the software without which my research would have been impossible. I particularly want to thanks all those who write documentation, manage websites, and do the other low-profile tasks that make it all work.

Chapter 1

Introduction

The use of dynamic languages, such as Python [62] and Ruby [67], has become more widespread over the past decade. There are many reasons for this, including ease of use and a greater use of programming languages by non-professional programmers such as biologists and web-designers. Whatever the reasons, it means that more and more computing power is devoted to running programs in these languages.

Increasing the performance of these programs could save considerable amounts of time and reduce energy consumption, especially as dynamic languages tend to perform relatively poorly compared with static languages, such as Java [42]. However, improving the performance of dynamic languages is difficult without considering how they are implemented.

Static compilation is inappropriate for dynamic languages, as the resulting executable would be very large and sometimes slower than the equivalent interpreted program, due to memory caching effects. Consequently, dynamic languages are implemented using virtual machines. Since virtual machines are a necessary feature of dynamic languages, improving the performance of dynamic languages requires improving the performance of the underlying virtual machines.

Knowledge about the efficient implementation of virtual machines for dynamic languages currently lags behind that for static languages, like Java and C#. Not only that, but the techniques used for implementing Java or C# are not necessarily the correct ones to apply to dynamic languages. Although implementation of virtual machines for dynamic languages is not as advanced as for static languages, improving the performance of dynamic languages is an active research area. For dynamic languages, such as Python, there are known techniques for increasing performance of their virtual machines.

As the optimisation of dynamic languages becomes more sophisticated, the engineering challenges in implementing them become greater. This is already a problem. Commonly used virtual machines use inefficient memory management

techniques, not because better techniques are not known, but because they are hard to implement. An important challenge for dynamic languages is how to handle the engineering aspects of implementing better virtual machines. The many components of a dynamic language virtual machine are not easily separated, and performance enhancing techniques can make this interweaving of concerns an impenetrable tangle. This is especially a problem for open-source or research projects, which often do not have the infrastructure required to use the heavy-weight software engineering techniques that might tame this complexity.

1.1 Virtual Machines

The term 'virtual machine' (VM) has a number of meanings. At its most general it means any machine where at least part of that machine is realised in software rather than hardware, but this is too broad a definition to be useful. In computer science, the virtual machine has come to acquire a number of related meanings; the book Virtual Machines [70] describes these. In this dissertation, the term virtual machine refers to a program that can execute other programs in a specific binary format, by emulating a machine for that program format. The custom binary format is usually known as 'bytecode' although, strictly, bytecode only refers to those formats where the instruction is encoded as a whole byte.

1.2 Dynamic Languages

The term 'dynamic language' is commonly used to refer to any language with dynamic typing. However, dynamic languages often have many features, beyond the type system, that static languages lack. For example, Python and Ruby include the ability to modify the behaviour of modules and classes at runtime, change the class of an object, add attributes to individual objects, and provide access to debugging features for running programs; the standard Python debugger is implemented in Python and can be imported and run by any Python program at run-time. For this reason, Python and Ruby are sometimes known as 'highly dynamic' languages. These highly dynamic languages are challenging to optimise, and thus their performance is generally somewhat slower than static languages.

Despite being slower, dynamic languages have a key advantage. Programs developed in a dynamic language tend to be shorter, and by implication, cost less to develop and have fewer defects. They also seem to be easier to learn and are popular among part-time programmers such as (non-computer) scientists and engineers.

This thesis is about the implementation of dynamic languages, particularly those languages supporting many dynamic features. Python was chosen as it is the most

widely used general-purpose highly dynamic language. PHP and Javascript are probably used more widely, but they are not as dynamic as Python nor are they really general-purpose languages, both being quite web specific.

1.3 The Problem

1.3.1 Developing VMs for Dynamic Languages

Developing a modern VM is a big project. Development of Sun's HotSpot Java Virtual Machine (JVM) [58] and Microsoft's Common Language Runtime (CLR) [55] have each taken a huge amount of resources. Other high-performance VMs for Javascript such as Tracemonkey [31] and the V8 engine for Google Chrome, also have large budgets and manpower resources when compared with community-developed or academic VMs.

This means that new or minority languages have either to run on unsophisticated VMs or be modified to work on a pre-existing platform such as the JVM. This can be a problem for dynamic languages, such as Python or Ruby. Although these languages can be made to run on the JVM or CLR, performance is relatively poor. For example, the Python implementations for the JVM and CLR are no faster than the standard Python implementation, CPython, despite the presence of a just-in-time compiler and high-performance garbage collectors [49, 41].

It is already too difficult for many open source or academic communities to produce a state-of-the art VM for a dynamic language. This situation will only get worse as new optimisations for dynamic languages are discovered; the engineering challenges of developing virtual machines for those languages will grow ever greater. The real challenge for making dynamic languages faster is not developing new optimisations, but developing new ways to build VMs that can incorporate those optimisations.

1.3.2 A Possible Solution

Although all VMs are different, some common features can be observed. All modern VMs interpret some sort of pseudo machine code, usually bytecode, and provide automatic memory management. It should be possible to hide these common features behind some sort of interface, either in the form of a tool or as a library. Specific VMs could then be specified using this interface. This would simplify the design of the VM as only the language specific parts would need to be considered.

1.4 Thesis

It is the central thesis of this dissertation that:

> The best way, in fact the only practical way, to build a high-performance virtual machine for a dynamic language is using a tool or toolkit.
>
> Such a toolkit should be designed around an abstract machine model.
>
> Such a toolkit can be constructed in a modular fashion, allowing each tool or component to use pre-existing tools or components.
>
> Using such a toolkit, it is possible to build a virtual machine that is at least as fast as virtual machines built using alternative techniques, and to do so with less effort.

1.5 Contributions

1.5.1 Core Contributions

This research :

> Demonstrates that a toolkit that uses ready-made components is capable of producing virtual machines with competitive performance.
>
> Describes an effective way to construct virtual machines for dynamic languages using a toolkit.
>
> Describes the how the optimisation techniques for dynamic languages differ from those for static languages and shows that those techniques are orthogonal to each other, specifically:
>
> that high performance can be achieved with standard compilation techniques by applying, at the bytecode level, optimisations specific to dynamic languages.

1.5.2 Ancillary Contributions

This research also:

> Describes a new extension of block-structured heaps, which allows pinning of objects and moving collectors to be combined within a generational framework.

Evaluates the relative costs and benefits of various implementation and optimisation techniques for Python, and by implication, other dynamic languages.

1.5.3 Software

Two pieces of software were produced as part of this research: the Glasgow Virtual Machine Toolkit (GVMT) and the HotPy Virtual Machine.

The Glasgow Virtual Machine Toolkit is a toolkit for building VMs for dynamic languages. High performance VMs can be constructed quickly using the GVMT.

The HotPy VM is an implementation of Python that can potentially serve as an experimental platform for VM implementation techniques. The HotPy VM supports all the core functionality of the language, but has limited library support.

1.6 Outline

Chapter 2 starts with a very brief history of VMs. The various aspects of VMs are then discussed, covering the following points: dispatching techniques available for interpreters; the relative merits of register-based and stack-based VMs; garbage collection techniques; and approaches to optimisation in VMs. The major VM implementations currently available are then surveyed. The chapter concludes by discussing the difficulty of implementing a VM incorporating all these many aspects.

Chapter 3 describes a new approach to constructing VMs. This approach consists of building a set of tools, or toolkit, based around an abstract machine model. The existence of the toolkit means that VMs can be designed without concern over difficulties of interfacing the various VM components. The abstract machine model allows the toolkit to be built in a modular fashion.

Chapter 4 describes the Glasgow Virtual Machine Toolkit, a toolkit based on the ideas from Chapter 3. It describes the abstract machine for the GVMT in detail. The tools in the toolkit are discussed, both front-end tools for converting source code to abstract machine code and back end tools, especially the just-in-time-compiler generator. An extension of block-structured heaps is described, along with a garbage collector which supports a copying collector and on-demand object pinning.

Chapter 5 describes the HotPy Virtual Machine, a VM for Python built with the GVMT. HotPy performs many optimisations as bytecode-to-bytecode transformations, as advocated in Chapter 3, separating the dynamic language optimisations from the low-level optimisations provided by the GVMT. HotPy is the first VM,

of which I am aware, that is designed around the use of bytecode-to-bytecode optimisations. The structure of HotPy is described, highlighting how the use of the GVMT influences the design. Emphasis is also laid on aspects of the design which differ considerably from the design of CPython.

Chapter 6 evaluates HotPy and to a lesser extent the GVMT. It shows that a toolkit can be used to construct a VM that compares favourably with the alternatives. By separating the various optimisations that HotPy uses, it is possible to show clearly that specialisation-based optimisations are more valuable for dynamic languages than traditional optimisations, and that purely interpreter-based optimisations can yield large speed-ups. It is also shown that specialisation-based and traditional optimisations are complementary; combining the two can yield very good performance.

Chapter 7 summarises the results and conclusions from the other chapters. It makes some suggestions for future work and outlines ways in which some of the results can be applied to existing VMs.

Appendices cover the full instructions sets of both the GVMT abstract machine and the HotPy virtual machine, as well results and

Chapter 2

Virtual Machines

As mentioned in the introduction, the term 'virtual machine' has a number of different meanings. In this chapter, and the rest of this thesis, the term virtual machine (VM) is taken to mean a program that directly executes a machine-readable program. The programs being run can be in text form, but are more usually in the form of pseudo machine-code. This chapter provides an overview of VM technologies, dynamic languages, and the relationship between the two.

2.1 A Little History

2.1.1 Early developments

The first virtual machine was, as far as the author is aware, the 'control routine' used to directly execute the intermediate language of Algol 60, as part of the Whetstone compiler, described in the Algol 60 Implementation [63]. The virtual machine of the Forth language [56] is the first virtual machine to be designed to be the primary, or only, means of executing a language.

The first bytecode[1] format to attain reasonably widespread use was the p-code of USCD Pascal [12]. P-code was loosely based on the o-code intermediate form of BCPL [64]. P-code was designed to be executed directly, was similar in form to real machine code, and could be compiled to machine code quite easily. Smalltalk was the first language to rely on a bytecode that embodied features not present in real machine codes, so in some sense Smalltalk bytecode was the first modern bytecode format.

The overhead of interpreting bytecode means that interpreted languages are almost always slower than native machine code. Consequently, compiling bytecode

[1] Some of the formats described are not strictly bytecode, but the term 'VM binary program' is rather cumbersome.

to machine code at runtime is an obvious performance improving technique, provided that the code is run a sufficient number of times to overcome the cost of compilation. The first runtime compilers were part of early LISP systems in the 1960s, but these created machine code directly from the abstract syntax tree. The Smalltalk-80 system included a just-in-time (JIT) compiler [24].

A more detailed overview of the field, including more history up to 2004, can be found in the two excellent overview papers: A Brief History of Just-In-Time [6] and A Survey of Adaptive Optimization in Virtual Machines [5].

2.1.2 Trends in Research into Virtual Machines

Until recently the only high-performance VM for a genuinely dynamic language was the Self VM [74]. Despite being more dynamic than its predecessor, Smalltalk, Self gave better performance, thanks to a sophisticated compiler. A number of novel optimisations were developed for Self [22], although a number of the more complex analyses were dropped in later versions, the type information being gathered at runtime instead [38].

The advent of Java shifted emphasis in virtual machine research from dynamic languages to static ones, and most research on virtual machines focused on the JVM and one JVM in particular, the Jikes RVM [43]. Over the last few years, research has again turned towards dynamic language VMs. This trend has been driven by the importance of Javascript for the world wide web and by the rise in popularity of 'scripting' languages, such as Python and Ruby.

2.1.3 Recent developments

The modern trend in bytecode-based languages has been towards expressiveness and utility over performance. This tends to mean that the individual bytecodes in languages such as Python and Ruby have a higher semantic level than languages like Java. These 'fat' bytecodes are harder to beneficially compile than the 'thin' bytecodes of Java, since the interpretative overhead is proportional smaller. Until quite recently, neither Python nor Ruby have had any runtime compilation capability[2].

There was little research into the efficient implementation of dynamic languages from the end of research into Self in the early 1990s until a resurgence in the late 2000s. The rise of Javascript and the increasing popularity of Python and Ruby has caused an increase in research into this area. Much of this recent research has been focused on optimisations determined dynamically rather than statically; see Section 2.4.3.

[2] The PyPy project(http://pypy.org), and Rubinius(http://rubini.us) added machine code generation capability to Python and Ruby VMs in 2009.

2.2 Interpreters

In computer science the term ‘interpreter’ is used to mean any piece of software that decodes and executes some form of program representation. This is taken to exclude the use of a physical machine ‘interpreting’ machine code. Although it is possible to interpret the original source code of a program directly, modern interpreters do not do so. They interpret some form of the program that has been translated into a machine-readable binary from the original human-readable textual source.

For the rest of this thesis the term ‘interpreter’ refers to a procedure that executes programs in a machine, rather than human, readable form (but not machine-code).

2.2.1 Interpreter dispatch techniques

In an interpreter, dispatch is the process of decoding the next instruction and transferring control to the machine code that will execute that instruction. Research on interpreter dispatch techniques has, unsurprisingly, been focused on improving the speed of interpreters. However, the speed of different dispatching techniques depends on the underlying hardware. As hardware design has changed over the years, particularly with the introduction of pipelining and super-scalar execution, so the relative performance of different techniques has altered.

Most modern interpreted languages are implemented by a two stage process where the source code is translated into code for a VM, then that VM code is executed by an interpreter. Although some interpreters, Perl5 and Ruby1.8, interpret a form that follows the original syntax, most use a form closer to the form of machine code.

Bytecode Dispatching

The most commonly used forms of VM code interpreter are Token Threaded and Switch Threaded. Figure 2.1 shows the pseudo machine code for Token Threading; the code to locate the address of the next instruction is duplicated at the end of every instruction. Figure 2.2 shows the pseudo machine code for Switch Threading; there is only one instance of the code to locate the address of the next instruction, `next`. All other instructions include a jump to `next`. The main advantage of these techniques is that the VM code is independent of the actual implementation. Switch Threading is so named because it can be implemented using the `switch` statement in C. Switch Threading has the advantage that it can be implemented portably in C (see Figure 2.3) but Token Threading is usually faster. For hardware that employs branch prediction, which is most modern hardware, the single dispatching point in the Switch Threading interpreter can cause

```
bytecode:        table:     push:                  add:
  1 /*push*/       &nop       *sp++ = *++ip          *sp++ = *--sp + *--sp
  A /*literal*/    &push      i = decode(*++ip)      i = decode(*++ip)
  1 /*push*/       &add       addr = table[i]        addr = table[i]
  B /*literal*/    ...        jump *addr             jump *addr
  2 /*add*/
```

Figure 2.1: Token Threaded Code – Stack-based program for A+B

more branch mis-predictions, making Token Threading significantly faster.

When the VM code is encoded in such a way that the first byte of each instruction contains only the token corresponding to the instruction, the code is generally known as 'bytecode'. When using bytecode the `decode` operation is not required, speeding up the dispatch. Bytecode is a very widely used form of VM code, being used in the JVM, CLR, Python, Ruby (1.9+), Smalltalk, Self and others.

```
bytecode:        next:                  table:
  1 /*push*/       i = decode(*++ip)      &nop
  A                addr = table[i]        &push
  1 /*push*/       jump *addr             &add
  B                                       ...
  2 /*add*/
                 push:
                   *sp++ = *++ip
                   jump next
```

Figure 2.2: Switch Threaded Code – Stack-based program for A+B

Address-Based Dispatching

An alternative to bytecode is to encode the program as a list of the addresses. Each address is the address of the code that implements the corresponding instruction. This form is known as Direct Threading, see Figure 2.4. Direct Threading [9] is original source of the term 'threading' in this context. The word 'threading' was used as the execution threads its way through the instruction stream and the interpreter machine code. Direct Threading was originally designed to reduce code size for compiled code, but the advent of larger memories made this use redundant.

A modified form of Direct Threading, which adds a level of indirection to the address fetching, is Indirect Threading; see Figure 2.5. Although Indirect Threading is slightly slower than Direct Threading, the extra level of indirection makes

```
next:
switch(*++ip) {
  case PUSH:
    *sp++ = *++ip;
    goto next;
  case ADD:
    *sp++ = *--sp + *--sp;
    goto next;
  case ...
}
```

Figure 2.3: Switch Threaded Implementation in C

```
thread:          push:                   add:
  &push            *sp++ = *(++ip)         *sp++ = *--sp + *--sp
  A              jump *++ip              jump *++ip
  &push
  B
  &add
```

Figure 2.4: Direct Threaded Code – Stack-based program for A+B

handling of data easier and is the standard threading method used in Forth implementations.

Another alternative is to encode the program as a series of calls. The encoding then becomes directly executable. Each instruction implementation would end with a return statement. This is Subroutine Threading; see Figure 2.6. Note that since the thread is executable code, data can no longer be embedded in the thread. Indirect, Direct and Subroutine threading were largely developed to keep programs small on machines with limited memory, rather than as techniques for implementing complex languages.

Before the advent of long pipelines in modern processors, Direct Threading gen-

```
thread:    push:                  add:
  &push      *sp++ = *(*ip+1)      *sp++ = *--sp + *--sp
  A           jump *(*++ip)         jump *(*++ip)
  &push
  B
  &add
```

Figure 2.5: Indirect Threaded Code – Stack-based program for A+B

```
thread:          push:              add:
  call push        *sp++ = *dp++      *sp++ = *--sp + *--sp
  call push        ret                ret
  call add

data:
  A
  B
```

Figure 2.6: Subroutine Threaded Code – Stack-based program for A+B

erally outperformed subroutine threading. However, modern pipelined processors do not handle VM instruction dispatching well, as the branches are hard for hardware to predict. For pipelined processors, Subroutine Threading makes the flow of control visible to the processor, which leads to fewer branch mis-predictions, and consequently better performance.

A variant on Direct Threading and Subroutine Threading is Call Threading. Call Threading encodes the program as a series of addresses, like Direct Threading, but performs calls, rather than jumps, to execute the instruction bodies. This has both the call overhead of Subroutine Threading and the poor branch prediction of Direct Threading and is thus the slowest of the three; see Figure 2.7. Call Threading, like Switch Threading, can be implemented as portable C.

```
loop:
 call(*ip++)
 goto loop

thread:          push:                add:
  &push           *sp++ = *(++ip)      *sp++ = *--sp + *--sp
  A               ret                  ret
  &push
  B
  &add
```

Figure 2.7: Call Threaded Code – Stack-based program for A+B

The fastest threading technique of all is Context Threading [11] which is an enhancement of Subroutine Threading that converts branches in instruction bodies directly into branches in the program thread. Performance can be further improved by inlining the bodies of some of the smaller instructions into the VM code.

It is debatable whether interpreters employing the fastest dispatching techniques, for example Context Threading with selective inlining, are really interpreters at

all. I would suggest that the line between interpretation and compilation has been crossed, and that the fastest interpreters are really just simple, easily portable, just-in-time compilers.

2.2.2 Register based VMs

Another approach to reducing the overhead of instruction dispatch is to reduce the number of instructions. This can be done by using a 'register' style instruction set.

Traditionally virtual machines have been implemented as stack machines. Both the JVM and the CLR are stack machines. Although stack machines are common in the VM world, hardware stack machines are extremely rare. The reason for this is that in hardware the stack is a bottleneck in data flow which makes it difficult for stack machines to compete with register machines.

However, the situation is different for a VM. In a software VM, the operands cannot be fetched and decoded in parallel. This means that stack machines do the same amount of computation as register machines; also stack machine code is more compact. The advantage of a register-based instruction set is that fewer instructions are required. Having fewer instructions increases performance of an interpreter, due to the reduced stalls caused by incorrect prediction of branches.

For a Pentium 4, Shi et al. [69] found an approximately 30% speedup for JVM code replacing a stack-based interpreter with a register-based one. However, the register based code was optimised to make more efficient use of the 'registers', but the stack code was not optimised to make more efficient use of the stack. Since Maierhofer and Ertl [52] found a speedup of about 10% from optimising stack code, this would suggest a reduced speedup of around 20%. It is worth noting that the above speedups where reported for a direct-threaded interpreter. As far as I am aware, there are no results available for a subroutine-threaded or context-threaded interpreter.

There are two mainstream VMs that are register based, the Lua virtual machine [40] and the Zend PHP engine. Lua switched from a stack-based bytecode to a register-based bytecode between versions 4 and 5. The implementers report speedups of between 3% and over 100% for a few simple benchmarks due to the change in instruction format. There is no stack-based equivalent to the Zend engine, so comparisons are not possible.

2.2.3 Compilation

Although the best performing interpreter (a register-based context-threaded interpreter) would be significantly faster than a simple stack-based switch-threaded

interpreter, it would appear that the overhead (both at runtime and in terms of engineering effort) would be better spent on genuine compilation. After all, a register-based context-threaded interpreter requires register allocation and the production of native code for branches and calls. It is only a short step to full compilation.

2.3 Garbage Collection

All major VM-based languages, with the exception of Forth, manage memory automatically. This makes the development of software much easier, although it does come at a small cost in performance. Automatic memory management is generally known as garbage collection, even though automatic memory management involves allocation of memory as well as collection of garbage.

Garbage collection allows languages and the programmers who use them to regard memory as an infinitely renewable resource. By tracking which chunks of memory are no longer accessible by the program, the garbage collector can recycle those chunks of memory for reuse. For the rest of this section, I will refer to these chunks of memory as 'objects', even though they may not be objects in the object-oriented sense.

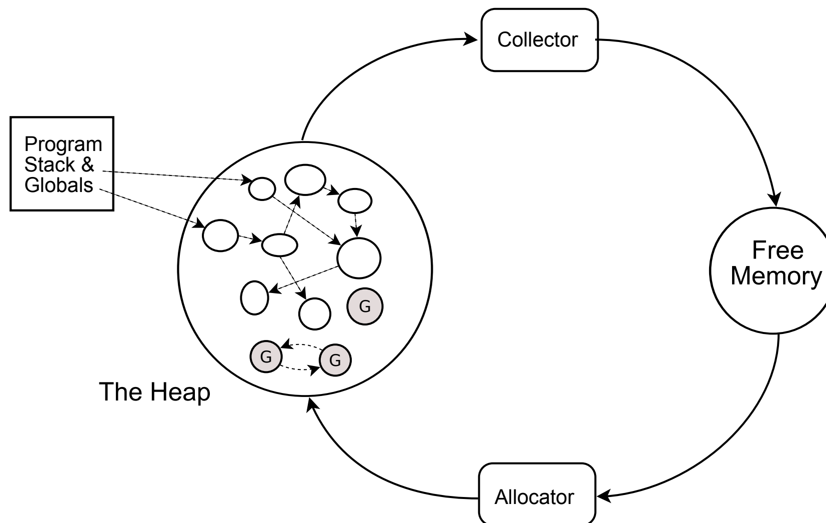


Figure 2.8: Memory Cycle

Garbage collection consists of two parts: an allocator which provides objects to the program, and a collector which recycles those objects that cannot be reached by the program. The collector reclaims objects so that the underlying memory can be freed and made available to the allocator. Figure 2.8 shows the memory cycle; the objects labelled `G` are unreachable and can thus be reclaimed by the collector; the memory they occupied is then free for use by the allocator.

While advanced collectors can run concurrently with the rest of the program, collections generally take place while the program is suspended. However, as the

number of processors on standard computers increases, concurrent collectors will probably become more common.

Since the design of collectors is considerably more complex than that of allocators, memory managers are generally described in terms of their collectors. Garbage collectors can be classified as either reference counting collectors or tracing collectors. 'Garbage Collection' [45] by Jones and Lins provides an excellent overview of the subject, although it is a little out of date. A more up to date list of publications can be found online at The Garbage Collection Bibliography maintained by Richard Jones[44].

Most research into garbage collection since 2000 has taken place using the MMTk [15] garbage collection framework in the Jikes RVM [3]. This has the advantage that various algorithms and techniques can be compared directly, but it does mean that it is rather biased towards Java applications.

Comparing the performance of various garbage collections is difficult as no one type of collector is faster than any other for all workloads. Despite this, it is possible to make some generalisations.

2.3.1 Allocators

Although much simpler than the collectors, allocators are an important part of a memory management system. Allocators come in two forms; free-list allocators and region-based allocators. Free-list allocators work by selecting a list that holds objects of the correct size (or larger), and returning the first object from that list. Region-based allocators work by incrementing a pointer into a region of free memory and returning the old value of that pointer. Region-based allocators are often called bump-pointer allocators, since allocation involves incrementing (or 'bumping') a pointer. Bump-pointer allocators are simple enough that their fast path can be inlined at the site of allocation, making them even faster. Obviously both allocators need fall-back mechanisms, either to handle empty lists in the case of a free-list allocator, or when the pointer would pass the end of the region in a region-based allocator.

Region-based allocators can allocate objects faster than free-list allocators. In general, only region-based collectors can free memory in a form suitable for region-based allocation.

2.3.2 Tracing

Most garbage collectors are 'tracing' collectors. Tracing collectors determine all live objects by tracing the links between objects in the heap. A collection is done by forming a closed set of all objects reachable, directly or indirectly through other

objects, from the stack and global variables. All remaining objects are therefore garbage and can be reclaimed. There are two fundamental tracing algorithms: copying and marking.

Copying algorithms move objects as they are found to a new area of memory. The entirety of the old memory area is then available for recycling. Copying collectors support region-base allocators. Marking algorithms mark objects as they are found. The unmarked spaces between marked objects are then available for recycling.

The cost of copying collection is proportional to the total size of the live objects. The cost of marking collection is proportional to the size of the heap, but with a significantly lower constant factor than for copying. So for sparse heaps (few live objects, lots of garbage) copying collectors are generally faster, whereas for dense heaps marking collectors are faster. In the real world, heaps tend to be neither sparse nor dense, but in the middle, so the choice and design of garbage collectors is not straightforward.

Marking Collectors

Marking collectors can be divided into three types: Mark and Sweep [54], Mark-Compact [10], and Mark-Region [17].

Mark and Sweep collectors are the simplest. After marking all live objects, all intervening dead objects are returned to the free list. Mark and Sweep collectors are prone to fragmentation and cannot be used with a region-based allocator.

Mark-Compact collectors avoid fragmentation, but are slower. After marking all live objects, all live objects are moved, usually retaining their relative position, to a contiguous region. The whole remaining space is thus unfragmented, allowing a region-based allocator to be used.

Mark-Region collectors reduce fragmentation and are of a similar speed to Mark and Sweep collectors. Mark-Regions sub-divide the heap into regions. During marking of live objects, both the object and the region containing the object are marked. Although only empty regions can be reclaimed, most memory can be reclaimed, since live objects tend to form clumps. To work well a Mark-Region collector needs a hierarchy of regions and must perform localised compaction. Localised compaction reduces fragmentation, but at a much lower cost than whole heap compaction. Mark-region collectors support region-based allocators.

It is possible to have very fast collection of regions by collecting an entire region at once, at a pre-determined point in the program [68]. However, this technique cannot be applied to dynamic languages as it requires extensive static analysis to determine when the region can be freed.

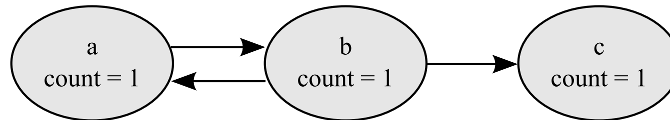


Figure 2.9: Uncollectable cycle for Reference Counting

2.3.3 Reference Counting

Reference counting garbage collectors work by maintaining a reference count for each object. This reference count for object `X` is the number of references to `X` from the stack, global variables and other objects. When this count reaches zero, the object may be reclaimed. Reference counting has two advantages. First, no separate collection phase is required, as collection is integrated with allocation. Second, as soon as an object becomes garbage, its memory is recycled.

However, reference counting also has two serious flaws. The first is that maintaining reference counts is expensive; reference counting garbage collectors generally have higher overheads than their tracing equivalents. The second is that if objects form a cycle, all reference counts remain above zero and cannot be reclaimed, even though the whole cycle is unreachable and thus garbage. Figure 2.3.3 shows a reference cycle that is garbage, but uncollectable by reference counting.

For interactive languages, the advantage of near-zero pause times for collection may outweigh the performance cost. Consequently there are a number of enhanced reference counting algorithms which handle cycles. Nonetheless the only widely used VM that uses reference counting is the CPython VM; all other Python VMs use tracing collectors. The CPython VM also includes an optional tracing collector, that collects cycles.

2.3.4 Generational Collectors

Generational collectors divide the heap into two or more regions called generations. Objects are allocated in the youngest generation, often known as a 'nursery'. If they survive long enough, they are promoted into the older generations over a series of collections. Generational Collectors generally give better perfor-

mance than simple collectors, if the rate at which objects become garbage differs for objects of different ages.

For most programs, what is known as the ‘weak generational hypothesis’ holds. The weak generational hypothesis states that young objects are more likely to die than older objects[3]. When the weak generational hypothesis holds, generational garbage collectors work well by collecting young objects frequently, which can be done cheaply, and collecting older objects infrequently.

In order to work correctly, generational garbage collectors must be able to find all live objects in younger generations. In order to be efficient, they must be able to do so without searching the older generations. This can be done by keeping a set of old-to-young references. The usual way to do this is to modify the interpreter to record any old-to-young references created in between collections. Generational collectors are generally faster than their non-generational equivalents, as the savings of not scanning the older generations outweigh the cost of maintaining the set of old-to-young references. However, it is not hard to construct a pathological program for which a generational collector is slower than the non-generational equivalent.

There is no requirement for a generational collector to use the same collection method for its younger generations as for older generations. In fact, it is common to use a copying collector for the nursery, which is usually sparse when collected, and a marking collector for the older generations, which are usually more dense. Having a copying collector for the nursery allows the use of a region-based allocator, providing fast allocation of new objects. It is also possible to combine a reference counting mature space and a tracing (usually copying) nursery. This should reduce pauses, as mature objects are collected incrementally.

2.3.5 Tagged Pointers

Values in dynamic languages are usually represented by a pointer to a block of memory which contains the value of the object. Representing a small value (such as an integer) by a pointer to heap object that contains that value is referred to a boxing; the object is a ‘box’ which holds the value.

Since objects in the heap will be aligned to some memory boundary (usually 4, 8 or 16 bytes) the low-order bits of a pointer will be zero. It is possible to represent a value directly in the pointer by ‘tagging’ the pointer. A simple tagging scheme for a 32 bit machine might be to store a 31 bit integer in the pointer by multiplying

[3]Note that the strong generational hypothesis, that states that objects become less likely to die as they get older, is not generally true. In other words, suppose that objects are divided in three ages, young, middle and old. The weak generational hypothesis states that young objects are more likely to die than middle or old objects, which is generally true. The strong generational hypothesis also states that middle objects are more likely to die than old objects, which is generally not the case.

30-bit signed integer	00

top 30 bits of word-aligned pointer	01

30 bits of IEEE floating point number	10

Garbage collector header bits	11

Figure 2.10: Self VM Tag Formats

it by two and adding one; setting the low-order bit to 1. Pointers would be left unchanged with the low-order bit set to 0. The value represented by the 32 bit word would be determined by examining its low-order bit; if the bit were set to 1 then the value would be an integer equal to the value of the machine word divided by two, otherwise the word would treated as a pointer. Figure 2.10 shows a more complex tagging scheme used by the Self VM.

2.3.6 Heap Layout

Many VMs are designed so that the whole heap is contiguous and the nursery and mature space are in fixed positions. This layout is simple to implement, and enables testing of whether an object is in the old or young generation by comparing its address with a fixed value. However, it is inflexible, only allows a fixed amount of memory to be used and might not work well with library code that uses its own memory management.

An alternative to a contiguous heap is to divide the heap into a number of blocks, or pages. It is possible to group objects of the same type onto the same page. The object's type can thus be determined from its its address alone, which allows a more compact representation of objects. This is know as a BIBOP (big bag of pages) implementation. The first documented used of this technique is for the MacLisp system by Steele [47].

Hudson et al.[39] describe a language-independent garbage collector toolkit which supports a heap divided into pages, and show how a generational collector can be supported. Each page has a generation, which allows the generation of an object to be determined quickly and without storing the information on a per-object basis.

Dybvig, Eby and Bruggeman [27] extend the BIBOP concept to what they call 'meta-type' information, which is simply any shared information about all objects

on a page, not necessarily their type. Their system provides a fast allocator, allocating all objects into the same page, then segregating them on promotion. Pages containing large objects are promoted from one generation to another without copying.

2.3.7 Garbage Collectors for Dynamic Languages

The interaction between the garbage collector, the rest of the program and the hardware is complex and very hard to analyse. It is thus almost impossible to determine the relative costs of various collectors except by direct experimentation.

Blackburn et al.[14] provide an empirical comparison of copying, mark-and-sweep and reference-counting garbage collectors, both generational and non-generational, for Java benchmarks running on the Jikes RVM. The results show wide variations in memory usage characteristics. This would seem to suggest that the relative performance of various garbage collection algorithms depends at least as much on the application domain than the language used. However, it is possible to make a few observations about memory management for dynamic languages.

Programs written in dynamic languages tend to obey the weak generational hypothesis, even if the same program written in a static language would not. This is because dynamic languages tend to allocate a large number of short-lived objects such as boxed numbers, frames and closures. Most of these extra objects are very short-lived, existing for the duration of a single function call or less.

Although optimisations can remove the allocation of many intermediate values, frames and closures [22], a considerable number will remain, and it would be surprising to find any sensible program written in a dynamic language that did not obey the weak generational hypothesis. This would seem to strongly suggest that the use of a region-based allocator and a copying collector for the youngest generation is almost mandatory.

Although it is desirable to use a copying collector, it may cause problems. Many dynamic languages, such as Python, Ruby and PHP amongst others, are expected to interact closely with libraries written in C. Interfacing libraries written in C with a garbage collector that moves objects can cause problems, as neither C compilers nor C programmers expect objects to be moved, seemingly at random. This does lead to the seemingly contradictory requirements that objects do not move, in order to interact with library code, and that objects can be moved by the garbage collector, for performance reasons. It should, however, be noted that only objects passed to library code need to be ‘pinned’; all others can be movable.

An alternative is to pass a ‘handle’ from the VM to the library code. This adds an extra level of indirection which may be unacceptable for performance reasons and in terms of complexity. For example, when passing large byte arrays from the VM and the I/O subsystem via a handle, it is necessary to either copy the whole

array or to access individual bytes via the handle. Both of these alternatives are expensive, so the ability to pin objects is highly desirable.

It would appear that a garbage collector for dynamic languages should be similar to a garbage collector for an object-oriented language like Java, with the requirements of very fast allocation for short-lived objects and the ability to pin objects. By using the BIBOP technique described in the previous section, pages can be pinned on demand; they can be promoted by changing the page tag. The Immix collector [17] supports pinning and region-based allocation, although it does not support a copying nursery. A design of segregated heap that builds on previous work and that supports both a copying nursery and on-demand pinning is described in Section 4.6.4.

2.4 Optimisation for Dynamic Languages

2.4.1 Adaptive Optimisation

Adaptive optimisation is a term used to describe optimisation that adapts to the running program. Adaptive optimisation focuses on spending optimisation effort where it will provide the greatest reward. This is done by comparing the estimated performance gain for optimising a piece of code with the cost of doing the optimisation; there is no point in optimising a piece of code that will only run once, whereas a piece of code that may run billions of times is worth optimising heavily. The idea of focusing optimisation effort on 'hot-spots' dates back to at least the 1980s [24]. The term adaptive compilation is often used instead.

Once code for optimisation has been selected, deciding which optimisations to apply to that code is something of an art. Most adaptive optimisers have a large number of tuning parameters which are set experimentally.

Optimisation control

Before code can be optimised, the optimisation controller must determine what code is worth optimising. There are two widely used approaches to optimising code. The first approach optimises code according to the static structure of the program, by optimising whole procedures or loops. The second approach optimises according to which code is actually used at runtime, determined dynamically by tracing the execution of the code. The former approach has been used for JIT compilation since the days of Lisp, and is still widely used, notably in the Sun HotSpot JVM, and for a statically typed language like Java gives very good results. The latter approach, that of optimising traces, is used in the TraceMonkey JavaScript engine of Mozilla Firefox, amongst others, and provides significant speedups for dynamic languages [31].

2.4.2 Whole Procedure Optimisation

The procedure based approach to adaptive optimisation records the number of times each procedure is called. Once the call count reaches a threshold value, then the procedure is optimised and compiled. In practice, various refinements are used. For example, the usage count can be made to decay, so that procedures must be executed frequently, not merely many times in total, before they are optimised. Another important refinement is to choose the procedure to be compiled by analysis of the call-stack, once a hot-spot has been reached. For example, it may be more profitable to compile the caller of the trigger procedure, rather than the trigger procedure itself, and thus be able to perform inlining.

2.4.3 Trace Optimisation

Trace[4] optimisation is a method of determining entirely dynamically which code to optimise. By taking advantage of the fact that optimisation will occur during the program's execution, tracing determines the code to be optimised by recording the actual execution of the program, with no regard to its static structure.

Traces must be selected before they can be optimised. Traces are identified by monitoring certain points in the program, usually backward branches, until one of these is executed enough times to trigger recording of a trace. During trace recording the program is executed according to the usual semantics, and the instructions executed are recorded.

Trace recording halts successfully when the starting point of the trace is again reached, but trace recording may not always be successful. One of the reasons for failure is that the trace becomes too long, but other reasons are possible; for example, an unmatched return instruction could be reached or an exception could be thrown.

If the trace completes successfully, then the recorded trace is optimised and compiled. The newly compiled code is then add to a cache. When the start of the trace is next encountered during interpretation, the compiled code can be executed instead.

During recording of a trace, branch instructions may be encountered, in which case the trace records the taken branch, and inserts a conditional exit at that point. During subsequent execution of a trace, the conditional exit may be taken. If this happens a sufficient number of times a new trace is recorded. These new traces need to be joined to the existing traces.

[4]Tracing in this context is completely separate from tracing in the garbage collection sense.

Trace Stitching

In the original trace-based optimiser, Dynamo[8], when a new trace is created, it is 'stitched' to the original trace. This is done by modifying the code at the exit point so that it jumps directly to the new trace. Trace stitching requires a cache of traces. When an exit from a branch is taken, the cache is checked. If an appropriate trace is in the cache it is stitched to the exit point.

Trace Trees

An alternative to trace stitching is to incorporate the new trace into the old one and re-optimise the extended trace. These extended traces are known as 'trace trees'[32] as the combined traces form a tree-like structure. For trace selection based entirely around loops, trace trees work well, but do rely on having a very fast compiler, since code may be recompiled several times.

2.4.4 Specialisation

Specialisation is a transformation which converts a general piece of code to a more specialised, and potentially faster, piece of code. Using specialised code also requires that guard code is inserted to prevent the new less-general code being executed when it would not be correct. In the event of a guard failing[5], and the specialised code being inapplicable, execution returns to the original code. Tracing can be viewed as a form of specialisation in which the code is specialised for the flow of execution through the program that is actually observed. Tracing can also drive additional specialisation by recording not only the flow of execution, but also the types of data. Code can then be specialised both for a particular path of execution and for the types of variable actually used.

2.5 Python Virtual Machines

Until the creation of Jython[49], there was only one implementation of Python, which served as the de facto specification for the language. There was no clear separation of language and implementation. Fortunately that situation has changed and the language is now reasonably well, if not formally, defined. Nonetheless the default implementation, now known as CPython, remains the reference standard.

All the Python VMs are under active development, often with the goal of actively improving performance. This, combined with the lack of standard benchmark

[5] Failure of a guard does not mean that it goes wrong, but that the condition it is testing is false.

suite, makes precise comparison difficult. Table 2.1 summarises the main Python implementations; the performance is from various developers own assessments, which seem to be in broad agreement with each other.

2.5.1 CPython

The standard implementation of Python, known as CPython as it is written in C, has evolved as Python has evolved. Consequently it still has a number of design features that, while perhaps appropriate for an early implementation of a new language, are not desirable in a modern, high-performance VM. These features are simple reference counting as a means of garbage collection, and a 'global interpreter lock', which prevents more than one interpreter thread executing at once. It is worth emphasising that these are not features of the Python language, merely the CPython implementation.

The choice of simple reference counting for garbage collection may have been a reasonable choice when Python was first evolving, but it is a real burden now. The global interpreter lock is an unfortunate side effect of the garbage collection strategy, as simple reference counting is not safe for concurrent execution. To make it safe would require extremely fine-grained locking, which would be prohibitively expensive for single-thread applications. So the global interpreter lock, which ensures only one thread is active in the interpreter at once, is used instead. The use of simple reference counting has a detrimental effect on CPython performance.

Implementation	GC	Threads	JIT	Performance (relative to CPython)
CPython	Reference Counting	G.I.L.	No	Same
CPython+Psyco	Reference Counting	G.I.L.	Yes	Faster (variable)
PyPy	Generational	G.I.L.	Yes	Faster
Unladen Swallow	Reference Counting	G.I.L.	Yes	Faster
Jython	As JVM	Native	Yes	Slower
IronPython	As .NET	Native	Yes	About equal

Table 2.1: Main Python Implementations

2.5.2 Psyco

Psyco[65] is a runtime compiler that interleaves, at a very fine level, interpretation, specialisation and compilation. It is extremely good at removing interpretative overhead as well as the overhead of having boxed integers and floating points. The speedups achieved vary from $\times 100$ for pure integer arithmetic code, down to $\times 1.1$ or less for some applications. The name 'Psyco' is a slightly jumbled acronym for the 'Python Specialising Compiler'. Although Psyco performs well

for some types of applications, it is somewhat ad hoc. The key ideas in Psyco were reused in the more robust and elegant PyPy project.

2.5.3 PyPy

The PyPy[66] project is two things in one: a translation tool for converting Python programs into efficient C equivalents, and a Python VM written in Python. The resulting VM executable is a translation of the PyPy VM source code, in Python, to machine code, by the translation tool. This means that the final VM has features present in the VM source code, plus features inserted by the toolkit. The tool is covered in more detail in Section 3.7.2. The PyPy VM implementation is fairly unremarkable (before translation) apart from the annotations to guide the translation process. The translation tool is responsible for inserting the garbage collector and generating the JIT compiler.

The PyPy generated JIT compiler uses a specialising tracing approach to optimisation. The tracing is done, not at the level of the program being executed, but at the level of the underlying interpreter.

Like CPython, the PyPy VM includes a global interpreter lock, which prevents real concurrency. However, it does not use reference counting for garbage collection, so it would be possible to make PyPy thread-capable by adding locking on key data structures. One of the PyPy developers, Maciej Fijalkowski, estimated that removing the global interpreter lock would be ‘a month or two’s’ work[29].

2.5.4 Jython and IronPython

Jython is a Python implementation for the Java Virtual Machine. IronPython [41] is a Python implementation for the .NET framework. The primary focus of each implementation is transparent interaction with the standard libraries for that platform; performance is a secondary goal. Both Jython and IronPython make use of their underlying platform’s garbage collectors and have no global interpreter lock. Both implementations need to make heavy use of locking, in order to be thread-safe.

Jython’s performance is poor compared to the standard CPython implementation. IronPython’s performance is better and is largely comparable with CPython, although as stated before, performance is not the primary goal for either implementation.

2.5.5 Unladen Swallow

Unladen Swallow is a branch of CPython which uses LLVM[50] to provide JIT compilation. It is a stated goal of the project not to do any new research, merely to implement already published optimisations. No attempts to remove the global interpreter lock or to implement better garbage collection are being made. The Unladen Swallow developers claim speedups ranging from $\times 1.1$ to $\times 1.8$ relative to CPython 2.6 for their benchmarks.

For a more detailed comparison of the performance of PyPy and Unladen Swallow see Section 6.4.5.

2.5.6 Static compilation of Python — ShedSkin

An alternative approach to improving Python performance is to translate Python programs to a statically-typed language. ShedSkin[26] is a Python to C++ translator. It uses type-inference to statically type whole programs, which can then be translated to C++. Unfortunately most Python programs are sufficiently dynamic that they cannot be statically typed. Since ShedSkin performs whole-program analysis, it must be able to type the whole program. For those programs which are amenable to this analysis, performance improvements are impressive.

Many programs written in dynamic languages are mainly, but not wholly, static in style. The problem is that a program that is 1% dynamic will cause ShedSkin to fail, whereas an adaptive optimising VM could give large performance gains.

For those programs that ShedSkin can handle, it gives an approximate upper bound for performance and a target for dynamic optimisers to aim for.

2.6 Other Interpreted Languages and their VMs

2.6.1 Java

The Java programming language[42] needs no introduction. Although it is a statically-typed language, the dynamic nature of class loading can present implementers of JVMs with some of the problems faced by implementers of high-performance VMs for dynamic languages. For this reason it is worth looking at implementations of Java and how they deal with the dynamic aspects of the language, especially as the techniques used have been covered in detail in a number of research papers and technical reports. Many of these techniques are applicable to dynamic languages.

Sun Hotspot

The HotSpot VM from Sun is the most widely available, and reference, implementation of Java. Its performance is good, it supports a number of platforms, and it is now open-source. HotSpot uses mixed-mode execution; it contains both an interpreter and compiler, and uses whole-procedure optimisation. HotSpot interprets code until it become 'hot' (hence the name), at which point the code is compiled. HotSpot uses whole procedure optimisation, rather than trace-based optimisation. The HotSpot compiler is a powerful optimising compiler; programs can be slow to start up, but long running programs can compete with C++ and Fortran for speed. Palenczny et al. [58] give a good overview, but most publications relating to it are more promotional than technical in style.

Jalapeño/Jikes RVM

The Jikes RVM (originally Jalapeño), from IBM, is a research VM implemented in Java. Unlike HotSpot, the implementation of the Jikes RVM is well described. The vast majority of papers written on optimising the JVM use the Jikes RVM as an experimental platform. The Jikes RVM is described in detail in an IBM technical report[3].

The Jikes RVM has no interpreter, but compiles all code on loading, quickly producing poor quality native code. It uses adaptive optimisation, optimising and recompiling code as necessary. So, unlike HotSpot, which has an interpreter and compiler, Jikes RVM has two compilers; a fast compiler and an optimising compiler. Like HotSpot, the Jikes RVM uses whole procedure optimisation. The approach used by the Jikes RVM is unlikely to be applicable unmodified to languages like Python, as most of the optimisation techniques are suited to static languages. Nonetheless, the basic premise of only optimising parts of the program which are most used is the fundamental idea behind high performance for bytecode-interpreted languages.

2.6.2 Self

The Self language[74] was developed from Smalltalk in the early 1990s. Self is a prototype-based, rather than a class-based, pure object-oriented language.

The Self Virtual Machine

The Self VM is described in Chambers' PhD thesis[22]. Chambers describes the various techniques used to reduce the overhead of the many dynamic features in Self. The Self VM described is significantly faster than the Smalltalk VMs that

preceded it, despite Self being more dynamic than Smalltalk. Chambers claimed to have achieved half the speed of equivalent C code, although most of the benchmarks were small and long running, reducing the effect of compilation time on total execution time. The Self VM is where many techniques used in modern JVMs and Javascript engines were first developed.

2.6.3 Lua

The Lua language was first developed in 1993. It is a dynamic language; variables are dynamically typed, but only a limited range of types are available. Its design goals, which have been adhered to throughout its development[40], are that the language should be simple, efficient, portable and lightweight. The authors define 'efficient' as not the same as fast, they define it as meaning 'fast while keeping the interpreter small and portable'. The standard Lua VM is a pure interpreter; no JIT compiler is included. Despite this, Lua is generally regarded as the fastest mainstream dynamic language.

There is an implementation of Lua with a JIT compiler, LuaJIT [51], which is faster still; the latest version has performance comparable with slower statically typed languages, such as Haskell.

2.6.4 Ruby

Ruby is an object-oriented dynamic language. Its motto is 'everything is an object'. Whilst it is similar to Perl in syntax, it is more a descendant of Smalltalk/Self than Perl.

Like Python, Ruby has a number of different implementations, but the default implementation is Ruby 1.8. Ruby 1.8 is unusual in not being a bytecode interpreter; the interpreter executes the abstract syntax tree directly. Also, like Python, Ruby has no official benchmark suite, and all implementations are under constant development. Table 2.2 summarises the main implementations; estimates of relative performance are intentional vague and may change.

Ruby 1.8 is also generally regarded as one of the slowest dynamic language implementations, and this is supported by benchmarks[23].

Like Python, Ruby also has implementations for the JVM (JRuby) and .NET (IronRuby). Benchmarking suggests that JRuby outperforms IronRuby, which contrasts with Python, where IronPython outperforms Jython. This would suggest that the JVM and .NET are roughly as good as each other for supporting dynamic languages; which is unsurprising since the JVM and .NET are fundamentally quite similar.

Ruby also has two other implementations, Ruby 1.9 and Rubinius. Ruby 1.9 uses a bytecode interpreter, and has performance loosely comparable to CPython. Rubinius aims to replace almost all of Ruby's standard library, which is currently written in C, with Ruby equivalents. In order to do this Rubinius must increase the performance of pure Ruby code considerably. Rubinius has largely achieved this goal thanks to a JIT compiler and more advanced garbage collection. Despite the more advanced internals, Rubinius is currently no faster than Ruby 1.8, as a result of having to execute libraries written in Ruby rather than in C.

Ruby's, like Python's, support for multiple threads of execution varies across implementations. JRuby and IronRuby use the underlying platform threads, and thus support threads well. Ruby 1.8 runs in a single native thread, performing switching of Ruby threads internally. Consequently only one Ruby thread can run at a time. Ruby 1.9 can support multiple native threads, but like CPython, has a global interpreter lock (Ruby 1.9 calls it a global VM lock), which prevents more than one thread executing bytecode at a time.

Ruby, the language, has features which presume the original implementation. For example, Ruby provides an iterator, `ObjectSpace::each_object`, which iterates over every object in the heap. Obviously, this causes problems for both garbage collection and concurrency. It makes using a moving garbage collector very difficult and causes problems for threads, as all objects are always globally accessible. JRuby has an option not to support this feature, as it causes performance problems on the JVM.

Implementation	GC	Threads	JIT	Performance (relative to 1.9)
Ruby 1.8	Mark & Sweep	Green	No	Slower
Ruby 1.9	Mark & Sweep	G.I.L.	No	Same
Rubinius	Mark-Region	Green	Yes	Slower (but improving)
JRuby	As JVM	Native	Yes	Faster
IronRuby	As .NET	Native	Yes	About equal

Table 2.2: Main Ruby Implementations

2.6.5 Perl

Perl was probably the first general purpose scripting language and is still widely used, although it is popularity is declining. Perl 5 is unusual in that the interpreter operates directly on the abstract syntax tree, rather than using bytecodes. It uses reference counting for garbage collection. The next version of Perl, Perl 6, uses a new VM, the Parrot VM.

The Parrot Virtual Machine

The Parrot VM [60] was designed to be a general purpose virtual machine for all dynamic languages. However, the only reasonably complete implementation of any mainstream language for Parrot is the Perl 6 implementation[6]. Parrot is a register-based virtual machine that includes, or is planned to include, pluggable precise garbage collection and JIT compilation. Exact details are hard to find and may change.

Performance data is also hard to come by, but the following may be indicative: in 2007, Mike Pall[7] posted his comparison of a few simple benchmarks comparing Lua running on the Parrot VM (version 0.4) with the standard Lua interpreter and LuaJIT[59]. Lua on Parrot was "20 to 30 times slower" than the standard Lua interpreter and "50 to 200 times slower" than LuaJIT. These numbers are not as bad as they may seem, as LuaJIT is very fast.

One would assume that performance had improved considerably since 2007, but in his blog of October 2009[76], Andrew Whitworth complained that for 'some benchmarks' the forthcoming release of Parrot, version 1.7, was 400% slower than the 0.9 release of January of that year.

2.6.6 PHP

PHP is very widely used in server-side web programming. The language is dynamically typed, but does not allow as much dynamism as Python. However, PHP supports a wide range of parameter passing and other complex features. The Zend PHP engine, which is the only widely used PHP engine, is unusual in a number of ways. Firstly it can be configured to use any one of three different threading techniques: call threading, direct threading or switch threading. The 'bytecodes' are in a VLIW[8] style and each instruction is very large (in the order of 100 bytes), including a machine address (for call threading or directthreading), operand indices, operand types and even symbol-table references. One of the more interesting features is that by using call threading or direct threading, the number of bytecode implementations can be essentially limitless, allowing the Zend engine to include large numbers of specialised instructions. Zend instruction operands can be of five types, and each instruction takes two operands; there are potentially 25 different specialisations of each operation. Zend has about 150 different opcodes. If all of these were to be specialised it would result in almost 4000 different instructions. This form of static specialisation is unique to the Zend engine, and is not applicable to object-oriented languages with an extensible type system.

[6] Even the Perl 6 implementation is not fully complete, but it is usable.

[7] Developer of LuaJIT

[8] Very Long Instruction Word.

2.6.7 Javascript

Javascript is probably the most widely executed interpreted language in existence, because of its use in web browsers; almost all smart phones as well desktop computers have one or more browsers, all with a Javascript interpreter. However, it is used rarely in any other environment so cannot really be regarded as a general purpose programming language. Javascript is a prototype-based object-oriented language, like Self, and many of the optimisations used in Self are applicable to Javascript.

Currently, the fastest Javascript engine is the V8 engine in Google Chrome. The V8 engine does not include a bytecode interpreter; it compiles the source code directly to machine code. This is a reasonable approach for Javascript, as the program is always delivered over the internet, never stored locally, so the overhead of parsing the source code and the generation of some form of code, whether bytecode or machine code, cannot be avoided. Simple machine code, with calls for complex operations, can be generated almost as quickly as bytecode. V8 uses a number of code optimisations from the original Self implementation. The two most notable are inline caches and maps. Maps, also known as hidden classes, record information about the layout of a particular object and are ideally shared by all objects with the same layout. Inline caches record the expected map of the receiver object at a call site, branching directly to the appropriate method if the actual map matches the expected map. V8 also includes a generational garbage collector, with a dual mark-and-sweep/mark-compact mature collector.

2.6.8 The Lisp Family

Lisp is a family of languages rather than a single language. The original Lisp was designed for symbolic computation and dates from the late 1950s[54]. Lisp and its variants are general purpose dynamic languages. The outstanding feature of Lisp is that all code can be treated as data. Syntax is very simple, allowing programs to be represented using the simple data structures (lists and trees) used throughout lisp programming. Manipulation of programs by themselves or other programs is relatively common place in Lisp programming.

The Lisp family has two main branches: Common Lisp and Scheme. There are number of differences between Common Lisp and Scheme. The most important difference, in the context of dynamic languages, is that Common Lisp includes optional type declarations. This means that Common Lisp VMs do not make much effort to optimise dynamically typed code. However, Scheme implementations must optimise dynamically typed code, if they are to perform well. Scheme also mandates that stack overflow will never occur as a result of using tail calls, whereas Common Lisp does not.

2.6.9 Scheme VMs

Scheme[72] is a version of Lisp with a standardised core and library. It has many implementations, two of which will be discussed here.

MzScheme

Mzscheme[9] is part of the PLT-Scheme[57] distribution and is a fast, mature Scheme interpreter with run-time compilation. Mzscheme is based around a bytecode interpreter. The abstract syntax tree is analysed and a number of optimisations performed before translating to bytecode. The JIT compiler is based on GNU Lightning[1]. The garbage collector is a partly-conservative collector derived from the Boehm collector.

Mzscheme has an unusual way of handling tail calls. Mzscheme converts simple tail recursion to loops in the bytecode, but all other calls use the C stack. When making a call that would overflow the C stack, the C stack is first saved to the heap, then execution jumps back up the stack, using the `longjmp` function. The function can then be called, as it will have sufficient stack space. Later, when the saved part of the C stack is required, it is restored. This approach allows Mzscheme to use the standard calling conventions of the underlying hardware, resulting in fast calls. Because the front-end converts tail recursion to loops, the stack saving mechanism should be only rarely required.

Bigloo

Bigloo[13] is a compiler for non-lazy functional languages, which emits C as its target language. Bigloo has a front-end for Scheme and ML. It uses a representation of the untyped lambda calculus, called Λ^n, as its intermediate representation. Bigloo includes a runtime evaluator, but it is not fully complete, nor designed for speed; thus Bigloo is not a strictly conformant Scheme implementation.

Bigloo translates Λ^n to C code, first performing many transformations on the Λ^n form. These transformations can be grouped into efficiency improving transformations and into transformations which transform the Λ^n code to a style better suited to translation to idiomatic C. The C compiler can produce better code from this idiomatic C than from C translated directly from lambda calculus. The code to implement these transformations runs to tens of thousands of lines of Scheme code.

Since, the Bigloo compiler can perform whole-program analysis it can perform many optimisations that would be impossible in an interactive system. The Bigloo runtime uses the Boehm conservative collector.

[9] mzscheme has been renamed ‘racket’ since the time of writing

2.7 Self-Interpreters

A self-interpreter is an implementation of an interpreter, or virtual machine, in the language being interpreted. It is possible to implement a language of sufficient power in itself trivially; for example, it is possible to implement a Python interpreter as follows:

```
import sys
execfile(sys.argv[1])
```

This sort of implementation is known as meta-circular evaluation. In order to really implement a virtual machine an implementation can only use features that can be directly translated to the underlying machine. The Jikes RVM, the Klein VM[75] for Self, and PyPy are all self-interpreters. The Jikes RVM and the Klein VM both include runtime compilers which can be used to bootstrap the VM. PyPy translates the running program to a lower-level form, usually C, which can then be compiled.

The perceived advantages of self-interpretation are that the VM can be written in a higher-level language and that library code can be more closely integrated with the VM since they are written in the same language. However, many components of a VM are quite low-level and writing them in too high-level a language may cause difficulties; code may involve a lot of 'magic' calls and be difficult to follow. In fact it may be better to write a VM in more than one language: a high-level language for high-level components and a lower-level language for lower-level components.

2.8 Multi-Threading and Dynamic Languages

The reader may have noticed that dynamic languages, particularly Python and Ruby, seem to struggle to support concurrency. Global interpreter locks are common implementations of these languages. So what is the problem?

The problem stems from the fact that these languages provide data structures, such as lists, sets and dictionaries, as fundamental types. Since these data structures are mutable, that is they can be modified, they require locking when used in a multi-threaded environment. Immutable data structures, such as tuples and strings do not need any locking.

Both Python and Ruby evolved in a single-threaded environment; machines with more than processor were rare. Python and Ruby programmers are accustomed to being able to write programs, even multi-threaded ones, without any synchronisation. This contrasts with, for example, Java programmers, who generally know

that synchronisation is required in multi-threaded programs which use mutable data structures.

The second problem is that these same data structures are heavily used internally in the implementations. The rest of this discussion will focus on Python, although similar arguments apply to Ruby. In Python, dictionaries are used internally to hold global variables and object instance variables. This could cause some unexpected interactions between threads. Suppose, for example, that one thread creates and stores a new global variable 'x' and another thread creates and stores a new global variable 'y'. In most languages, one would expect that (at least at some future time) both 'x' and 'y' would exist and be visible to both threads. In a multi-threaded Python without synchronisation, it is possible that 'x' (or 'y') would not exist at all. What would happen is that the dictionary holding the global variables might need resizing to insert a new variable. Both threads would then attempt this resizing at the same time, inserting 'x' and 'y' respectively into their local copy. Both then write back the new dictionary at the same time, a race condition, and one or other of the modifications is lost. Similar problems might occur with the dictionaries used to hold instance member values.

There are a number of possible solutions to these problems, which range between the following two extremes:

1. Design all built-in, mutable data-structures so that they are fully thread-safe. That is, use locking for all operations on these data-structures. This is potentially very expensive.

2. Insert the absolute minimum number of locks to ensure that the integrity of the VM is not compromised. The amount of locking required is that which would prevent the VM crashing. This would put the responsibility for locking objects on the programmer in a similar way to Java.

IronPython and Jython both use solutions similar to 1 above.

2.9 Conclusion

High-level, dynamic languages are more popular than ever. Despite this, the quality of implementation seems not to have improved over the last 15 years, although there are some improvements with the current (2010) generation of Javascript engines. The reasons for this become clearer when one considers that most of the research on improving VM performance was done on a language, Self, with a very simple VM; the Self VM had 8 bytecodes. Python has about 100. The engineering effort to implement the Self VM, writing the entire VM from scratch, was large. For a language like Python the effort would be enormous and beyond the means of most organisations. A different approach to building VMs is required.

Chapter 3

Abstract Machine Based Toolkits

This chapter describes a method of constructing virtual machines using a toolkit designed around an abstract machine model. In this chapter, the term 'abstract machine' is defined, and an abstract machine model is outlined which incorporates essential features of a VM for dynamic languages. The requirements for a toolkit for constructing VMs are discussed and the components of such a toolkit are outlined. The design and implementation of such a toolkit is justified as it reduces overall complexity, but does not limit the developer's ability to construct a high-performance VM.

3.1 Introduction

Development of a high-performance VM is no easy task, especially for the complex VMs required for dynamic languages. Although some components of a VM can be designed and implemented separately, others are bound together quite tightly. For example, in order to use a precise garbage collector, all code that manipulates pointers into the heap must be identified. These pointer manipulations may be in library code, in the interpreter or even in code that has been generated at runtime. For an evolving language like Ruby or Python, all the components must conform to the new semantics whenever a change occurs. This is especially an issue for JIT compilers; whenever a new bytecode is added, or the semantics of existing bytecodes change, the compiler must mimic the changes in the interpreter exactly.

Unless some way is found to reduce this complexity in the interactions between the components, the creation of new VMs will be possible only for large organisations. This would be a real loss both for academia, in terms of creating new experimental languages, and for languages supported by community development such as Python and Ruby.

By separating the parts shared by many VMs, from the language specific parts, the construction of a VM can be simplified.

3.1.1 Abstract Machines

Put simply, an abstract machine is a machine definition, rather than an implementation. The terms 'abstract machine' and 'virtual machine' are both used to describe some sort of intermediate representation between a source language and a target machine, usually a hardware machine. It is important to differentiate between abstract machines and virtual machines, at least for the purpose of this thesis. Unfortunately, the terms are commonly used interchangeably.

Although the usage of the two terms is similar, it is possible to observe some differences in general. The term 'abstract machine' is generally used when the machine language is used as a translation step between two other languages. For example the 'abstract continuations machine'[4] and the Spineless-Tagless G-Machine[46] are both described as abstract machines, and are used as an intermediate representation. The term 'virtual machine' is more often used when the machine language is evaluated directly. For example, the JVM and CLR are usually referred to as virtual machines. The distinction is important as virtual machine languages are designed for execution, whereas abstract machine languages are designed for translation into an executable form.

For the purposes of this thesis, an abstract machine language is textual and is designed to be translated into another form, whereas a virtual machine language is binary and is designed to be executed directly.

The first well-defined abstract machine was probably the intermediate language for Algol 60, mentioned in Section 2.1.1. Diehl, Hartel and Sestoft[25] list a large number of abstract machines and virtual machines, using the term abstract machines for both, regarding a virtual machine as an executable abstract machine.

3.1.2 A Toolkit for Constructing VMs

One approach to building VMs is to construct a set of tools, or toolkit, to build a VM. Such a toolkit would build a VM from a specification of the interpreter and supporting code. This approach is embodied in both PyPy and the Glasgow Virtual Machine Toolkit (GVMT), which is described in Chapter 4. Both these toolkits are able to generate a VM with JIT compiler and integrate a precise garbage collector, from a specification of the interpreter and supporting code. Section 4.9 includes a detailed comparison of the GVMT and PyPy.

The great advantage of a VM development tool, or toolkit, is that many parts of the VM can be handled by the toolkit. Generic features of the VM, such as a garbage

collected heap, can be conceptually separated from the VM specification details, such as the semantics of bytecodes, data representation and supporting functions. This leaves the developer free to deal with the language-specific parts in any way they choose, thus speeding development with little or no loss in flexibility.

Once such a VM development toolkit has been created, new VMs can be easily constructed that support advanced garbage collection and just-in-time compilation; the developer just needs to specify the bytecode interpreter and write any supporting code.

3.2 The Essential Features of a Virtual Machine

In order to decide what features a toolkit should support, it is useful to examine what features are common in modern VMs.

3.2.1 Garbage Collection

Garbage collection is a common feature amongst VMs[1]. Efficient garbage collection is, along with JIT compilation, one of the keys to good VM performance.

Although, as discussed in Section 2.3.7, a garbage collector for a dynamic language has a number of specific requirements, the performance characteristics of the garbage collector need not be part of the abstract machine. It is necessary only that the abstract machine supports garbage collection. An important point to note about garbage collection is how pervasive is its effect on the generated code. All code, whether in the interpreter, in JIT compiled code or in supporting code, must be implemented in such a way that all references to objects can be found and modified by the garbage collector. This means that all roots, that is pointers from the stack or global variables, into the heap must be identifiable.

3.2.2 Execution Control

Control of the execution, or flow, of a program is a key part of any language. Execution control can be divided into two types: concurrent and serial.

Control over the concurrent execution of a program is, at the operating-system level, either by processes or threads. Processes are quite loosely coupled, communicating only by messages. Threads are more tightly coupled, sharing memory. Since both processes and threads are provided by the operating system, the abstract machine needs to interface cleanly with these features, but it does not need

[1] A notable exception is the Forth VM.

to provide them.

The control over a single thread of execution varies widely between languages. As well as simple flow control in the form of branches and subroutines, modern languages provide non-local transfers of control in the form of exceptions, co-routines or continuations. Continuations are the most powerful of these, and capture most of the execution state of a program at the point at which they are created. It is possible to implement both exceptions and co-routines with continuations, but continuations require significantly more resources than either exceptions or co-routines.

In addition, other forms of flow control are conceivable and it should be possible to implement new ones on the abstract machine.

3.3 An Abstract Machine for Virtual Machines

An abstract machine is usually designed with one programming language in mind, but could easily be reused for other languages with similar semantics. For example, the 'abstract continuations machine' was designed for use in compiling ML. However, it has no ML specific features in it; it would support any non-lazy functional programming language that required support for continuations. The Spineless-Tagless G-Machine, designed for Haskell, should be able to support Miranda or another lazy functional programming language. Even the Warren Abstract Machine, designed for Prolog, might be a good target for any language requiring some sort of backtracking.

Although these three abstract machines were designed to implement a specific language, the machine specifications can be defined in terms unrelated to the source language definitions. Theoretically, any Turing complete abstract machine could act as a target for any language. However, if the language requires a feature in order to run efficiently and the abstract machine does not support that feature then it will be difficult to make an efficient implementation. Likewise, if an abstract machine is designed to support a feature that the language does not need, the overhead of the unused feature may impact performance.

Requirements for the Abstract Machine

The abstract machine for a virtual machine should serve as an intermediate representation between the language used to define the VM and the hardware. The semantic level of the abstract machine will therefore lie between that of the hardware and the virtual machine.

This means that the abstract machine should have a language that is a suitable target for a C compiler or similar, and should provide features required for con-

structing a dynamic language VM. To be a suitable target, an abstract machine language must have an comprehensive instruction set with well-defined semantics. The abstract machine language is not expected to be executed without further translation.

It is necessary to be able to compile the abstract machine code into machine code for a real machine, and to do so reasonably efficiently. The abstract machine can be viewed as the intermediate representation between source code for the VM and machine-code implementation of that VM. Like a compiler's intermediate representation it should be designed to be an effective bridge between the source code and the final output, which in this case is a VM. The abstract machine should support features common to VMs, without constraining the design of individual VMs unduly.

A VM Toolkit Based on an Abstract Machine

In order for a toolkit to translate its input to actual machine code, it is necessary to define the semantics. By using an abstract machine as a form of intermediate representation, the semantics can be defined in terms of the abstract machine, and all tools can easily collaborate to form a usable toolkit. Implementation of the toolkit is simplified as tools can be separated into front end (source to abstract-machine code) and back end (abstract-machine code to real-machine code) components.

Developing an abstract machine for VMs can simplify the development of VMs by separating the development into two parts: developing the tools to translate VMs described in terms of the abstract machine into executable programs; and design and development of the VM itself. The tools can potentially be reused for other VMs. By choosing the design of the abstract machine so that it separates the parts which are general to all VMs from the parts which are specific a particular VM, the overall development effort can be reduced significantly.

3.3.1 Designing an Abstract Machine

While an abstract machine should be as general as possible, it will have to be tailored to its intended domain to some degree. For example, will the abstract machine be stack-based or register-based? How much control over memory management should the VM developer have? What sort of support for runtime optimisations will the abstract machine provide?

A balance needs to be found between specificity and generality. An abstract machine should not be so specific that it only supports one VM, neither should it be so general that it is no more useful for building VMs than a standard compiler. The design space can be viewed as a spectrum running from language-specific VMs, such as the JVM, to general purpose compilers, such as the GNU C/C++

compiler. A useful abstract machine should lie somewhere between these two extremes. Some features of VMs, such as garbage collection, are almost universal, so it is obvious that the abstract machine should incorporate them. Others features, such as continuations, are much less common and it is a matter of judgement as to whether they should be included.

3.3.2 Special Status of Interpreters

The concept of the interpreter as a special entity is key to building VMs using a toolkit. The interpreter is not just another function. Because the VM is a program that runs programs, the abstract machine must support not only the program, the VM, but the program run on the program, the bytecodes. To do this, interpreters need to be treated specially. The state of the interpreter represents the execution state of the interpreted program, which needs to be supported by the abstract machine.

By differentiating between interpreters and other functions, the semantics of compiler and interpreter generators can be more clearly stated. This special status makes it much simpler to define and generate a compiler which guarantees that the behaviour of compiled code *exactly* matches that of the interpreted original. The interpreter's instruction pointer becomes part of the abstract machine state, on a par with the hardware machine's instruction pointer, enabling a unified approach to handling execution control in both the interpreter and supporting code.

Treating the interpreter as a special entity also has advantages for efficient implementation. Since the interpreter is part of the abstract model, the interpreter should integrate seamlessly with the rest of the VM. When implemented, calling from interpreted into compiled code, or vice-versa, should cost no more than any other machine-level call.

3.3.3 Compilation

If a VM is to achieve good performance it needs a JIT compiler to convert sequences of bytecodes to machine code; a key feature of a toolkit for building VMs is to generate that compiler. The toolkit should be able to automatically generate a JIT compiler from the interpreter specification.

An automatically generated compiler should produce code that has exactly the same semantics as the bytecodes it derives from. Of course 'exactly the same semantics' will depend on the exact abstract machine but a reasonable interpretation is that the observable behaviour should be the same.

Put formally:
Given a compiler generator CG and an interpreter generator IG, provided by the

toolkit, and a set of bytecode definitions B_{vm} provided by the VM developer. Then, during VM construction, the interpreter I_{vm} and compiler C_{vm} are generated as follows:
Interpreter Generation: $I_{vm} := IG(B_{vm})$
Compiler Generation: $C_{vm} := CG(B_{vm})$
At runtime, when the VM is executing, given some valid bytecodes b and an input x:
Bytecodes b can be compiled with C_{vm} to produce c^b_{vm}:
Compilation: $c^b_{vm} := C_{vm}(b)$.
When the compiled code is executed with input x, it should be equivalent to interpreting b with the interpreter I_{vm} and input x. That is:
$c^b_{vm}(x) \equiv I_v m(b,x)) \ \ \forall b,x$, provided that b and x are valid. What values of b and x are valid depends on both the set of bytecode definitions B_{vm} and the abstract machine definition.

Correctness

One of the main reasons for using a toolkit is the ability to specify the interpreter and the compiler from a common source. It is important that the interpreter and compiled code are effectively equivalent. Confidence that this is the case can derive from formal proof of the equivalence or statistical evidence in the form of testing.

Formal verification of the toolkit will be impossible unless it is possible to verify all the components. In order to reduce the engineering effort required to create a toolkit, reuse of external components such as the C compiler or a runtime compilation library is necessary. Proving the properties of these components is not feasible.

Since formal verification is impractical, validation must be done by testing, code reviews and other software engineering techniques. Although the use of these techniques is outside the scope of this thesis, it is worth noting that the use of external components removes certain categories of errors, as these components must verify their input to some extent and can be trusted to generate correct output for the given input. For example, if the interpreter generator uses the C compiler to generate machine code, then some classes of low-level errors in the final machine code, such as the use of incorrect calling conventions, will not occur.

3.3.4 Introspection

Introspection is the ability of a program to examine and perhaps modify the state of the underlying machine, which in this case is the abstract machine. The full state of the abstract machine should be visible to the program, although efficiency requirements may mean that only parts of it are modifiable.

Introspection is useful for a couple of reasons. It allows the VM to provide support for debugging and tools. It is also useful for supporting advanced language features. For example, continuations can be created by using a combination of non-local jumps, for flow control, and using introspection features to record necessary stack and heap information.

3.4 Optimisation in VMs for Dynamic Languages

One of the most important measures of a virtual machine is its speed. A VM that includes the ability to optimise code at runtime will almost invariably be faster than one that does not.

Optimisations can be loosely grouped into lower-level traditional optimisations, used in conventional compilers for static languages, and higher-level optimisations which are often language-specific. There must also be an intermediate representation which allows the two levels of optimisation to communicate. For a dynamic language, the main aim of these high-level optimisations will be, in general, to remove as much dynamism as possible, thus producing intermediate code that traditional optimisations can turn into efficient machine code.

3.4.1 Traditional Optimisations

Although dynamic languages may require new and interesting optimisation techniques, they also require traditional compiler techniques in order to provide good performance. These techniques include sophisticated register allocation, constant propagation and other optimisations found in most compiler textbooks. These optimisations can be applied after the high-level optimisations, so standard tools can be used.

3.4.2 Intermediate Representations

To translate from a high-level representation, such as bytecode or source code, directly to machine code is almost impossible to do well. By using one or more intermediate representations, the translation can be made much simpler and more effective.

A wide range of intermediate forms are possible. These intermediate representations can be loosely classified as either program-level or machine-level. Program-level forms contain all, or most, of the semantic information present in the original program and are suitable for language-specific optimisations. Machine-level forms are suitable for traditional optimisations and code generation. Lower-level

languages generally do not need program-level representations. For example the GNU C/C++ compiler (GCC) has two intermediate representations, GIMPLE[2] and RTL[3]; both can be considered to be machine-level representations.

Effective optimisation of dynamic languages requires different sorts of optimisations from C and a program-level intermediate representation is needed. Most dynamic languages already have a program-level representation, their bytecode. Bytecode is a very flexible format, carries program level information, and makes a good intermediate representation.

3.4.3 Adaptive Optimisation Engines

As discussed in Section 2.4.1, an adaptive optimisation engine consists of a controller, which selects code to optimise, and an optimiser, which transforms the code. The term 'adaptive' is used as the engine adapts to the running program; its behaviour is determined at runtime.

Once a sequence of bytecodes has been selected for optimisation, whether it is a whole procedure or not, it is first translated into a program-level representation, then high-level optimisations are applied to it. Then it is translated into machine-level representation, low-level optimisations are applied, and finally it is translated into machine-code.

Obviously additional stages can be added or stages omitted, but this idealised model will serve as a useful reference point. Figure 3.1 shows a generalised translation path from bytecode to machine code.

3.4.4 Building an Adaptive Optimisation Engine Using a Toolkit

There are two parts to building an optimisation engine. The first part is the selection of the code to optimise. The second part is the optimisation itself. The selection of code is largely language specific and should not require direct support from the abstract machine or toolkit. The second part is not only more complex, but is strongly influenced by the design of the abstract machine.

In Figure 3.1, it should be noted that the first two translation steps, from bytecode to optimised high-level intermediate representation (IR), are largely language specific and unrelated to the abstract machine, whereas the later steps are more abstract-machine specific.

As mentioned in Section 3.4.2, bytecode can fulfil the role of a program-level

[2]Generic structured InterMediate rePresentaion LanguagE
[3]Register Transfer Language

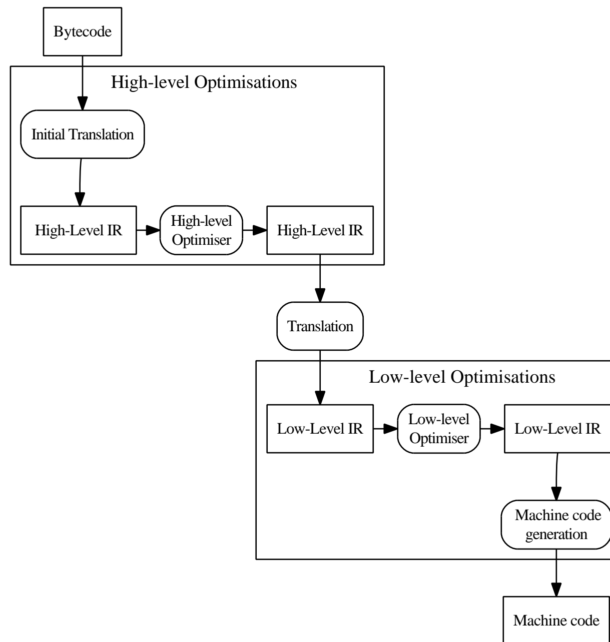


Figure 3.1: Generalised Bytecode Optimiser

IR; it is simple to analyse, and usually contains all of the semantic information present in the source code. Figure 3.2 shows an optimisation path using bytecode as a program-level IR, including toolkit generated components to translate from bytecode to machine code.

In order to ease the construction of bytecode-to-bytecode translators, the toolkit should support creation of arbitrary interpreters over the same bytecode used for the main interpreter. As an additional benefit, this will also ease the creation of bytecode disassemblers, verifiers, and similar tools.

The abstract machine specifies neither the means of optimisation control nor any language-specific optimisations. Whilst this may seem to be an omission, it allows the VM developer to choose an appropriate overall design, and not worry about the lower-level details.

3.5 When to use the abstract machine approach?

One question that has not been directly addressed so far, is this: Is the abstract machine approach worth using for a single VM; in other words, is it worth constructing a toolkit such as the GVMT just to create a single VM? The answer depends on the complexity of the resulting VM. For a very simple or toy VM, the answer must be no, but for a VM for a complex language like Python, the answer is probably yes. The ability to add and remove bytecodes easily, and to be able to develop the garbage collector separately from the rest of the VM, yet have it well

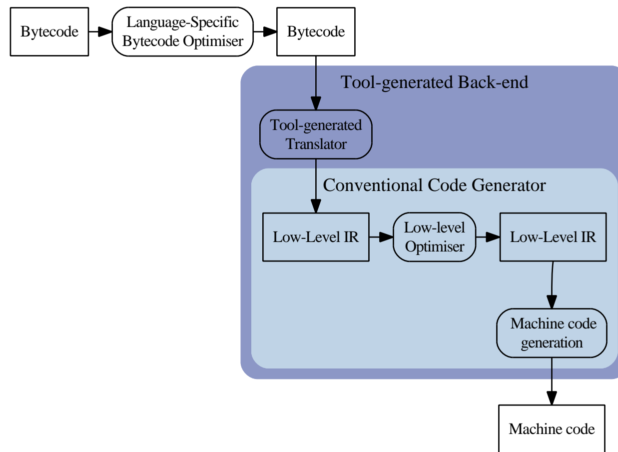


Figure 3.2: Toolkit Assisted Optimiser

integrated for performance benefits, is very productive.

If a toolkit already exists, it is worth using even for a small or prototype language, as using that toolkit should produce a better VM than using a pre-existing VM such as the JVM; this is demonstrated in Section 4.10.

3.6 Alternative Approaches to Building VMs

There are many possible ways of developing VMs, but four categories cover all existing and proposed approaches: creating a tool to build VMs; building a truly general-purpose VM; assembling a VM from a library of components; and building an adaptable VM. The first of these has already been covered in some detail.

3.6.1 A General Purpose VM

One approach would be to build a genuinely general purpose VM, that is, a VM with an instruction set so broad that a very large range of source languages could be translated to it. The only attempt to do this, of which I am aware, is the Parrot VM which was discussed in Section 2.6.5. The problem with this approach is that the VM must support many features which will not be required for any given language, but will still add overhead.

3.6.2 A Component Based VM

Another approach would be to build a library of common components. While this has been done for memory management[15], creating a library of compilers for

all possible bytecodes is clearly impossible.

It is possible to build a VM from components, provided the translation from bytecode to lower-level representation is coded manually. VMKit[33] is a JVM built using LLVM as JIT compiler back-end, the Boehm collector[18] for memory management and GNU classpath[34] to provide the libraries. The resulting VM is a compiler-only design, like the Jikes RVM. It has competitive performance once running, but is very slow to start up and has relatively poor performance for memory intensive applications. The slow start up is as a result of having to compile significant amounts of library code at runtime. The poor performance for memory intensive applications is due to the use of a conservative garbage collector.

3.6.3 An Adaptable Virtual Machine

A third approach is to build an flexible VM that can be adapted to suit new languages dynamically. This could either be a general-purpose VM that is then trimmed down, or a minimal VM with the ability to add new bytecode instructions at runtime. The latter approach is taken by the MVM/JnJVM project[73]. The MVM is an extensible VM with a small instruction set, supporting JIT compilation and garbage collection. The instruction set can be dynamically extended by loading new capabilities defined in a Lisp-like language.

A Java Virtual Machine, JnJVM, is created by modifying, at runtime, the MVM. Execution happens in two phases; the first phase is loading the new bytecodes, which extends the MVM; the second phase runs the bytecode program in the new, extended, VM. The current approach of creating the VM at run time would probably be unacceptable when running small scripts, although it would probably be straightforward to do the adaptation at build time.

This would appear to be a promising approach, but it is not clear how far from the core MVM the VM could be extended and still perform well. Unfortunately, research in this direction seems to have ceased.

3.7 Related Work

3.7.1 Vmgen and Tiger

Vmgen[28] is the interpreter generator used to build the GForth VM. Vmgen is focused on producing fast interpreters, and can produce very fast interpreters for a number of different architectures. However, vmgen does not have the ability to produce a compiler, nor does it support easy integration with the other components of the VM. A more sophisticated version of vmgen, Tiger[21], is available, which

is designed to further enhance interpreter performance and ease of use, rather than adding other tools.

3.7.2 PyPy

The PyPy project[66] consists of two components: a translation tool for converting interpreters written in RPython (a slightly restricted form of Python) into VMs; and a Python interpreter written in RPython. The translation tool can translate any RPython program into reasonably efficient C (and other statically-typed representations), although its primary purpose is to compile the Python interpreter.

The PyPy translation tool, termed 'translation tool-chain', converts the high-level (RPython) representation to successively lower-level representations, by using whole program analysis to remove the dynamism inherent in (R)Python. The JIT compiler generated by PyPy works by tracing the execution of the interpreter[19], rather than the execution of the program[4]. The interpreter source is annotated in order to help the compiler generator determine what to compile and when.

A detailed comparison between PyPy and the GVMT can be found in Section 4.9.

3.8 Conclusions

A VM consists of a number of tightly coupled components. Although these components cannot be developed independently, the creation of a number of these components can be automated, freeing the developer to concentrate on higher-level issues. By designing a low-level abstract machine and developing an accompanying toolkit, aspects such as memory management and JIT compilation can be greatly simplified, allowing the VM developer to concentrate on issues such as optimisation policy or whatever novel features the new VM includes.

In their paper on VM construction for dynamic languages, Bolz and Rigo[20] conclude that writing VMs 'by hand' is unsustainable and that some sort of tool(s) are required. Although the mechanism of automation in PyPy differs from the GVMT, automation is a necessity. A toolkit that implements the features discussed in this Chapter is presented in Chapter 4.

[4]PyPy initially aimed to support runtime compilation using partial evaluation, although attempts to do this have now been abandoned.

Chapter 4

The Glasgow Virtual Machine Toolkit

The Glasgow Virtual Machine Toolkit (GVMT) is an embodiment of the abstract machine principle discussed in Chapter 3. The GVMT is designed to support construction of dynamic languages. A manual for the GVMT is available from `http://code.google.com/p/gvmt/downloads/list/`.

In this chapter I will give an overview of the GVMT and describe some of its novel features in more detail.

4.1 Overview

The GVMT is based around an abstract machine definition and consists of two sets of tools: front-end tools to convert C source code to abstract machine code; and back-end tools to convert the abstract machine code into a working virtual machine.

The front-end tools consist of a C compiler, an interpreter generator and secondary-interpreter generator. The C compiler converts C code into instructions for the abstract machine. The interpreter generator and secondary-interpreter generator convert C-style interpreter definitions into instructions for the same abstract machine.

The back-end tools are a compiler-generator, to generate a compiler from the abstract machine bytecode specification, an assembler to convert abstract machine code to machine code, and a linker to ensure that components are laid out in a way that the garbage collector can understand. Figure 4.1 shows how the tools are used to generate an executable via the abstract machine code.

The GVMT abstract machine specifically targets dynamic languages. It is a stack-

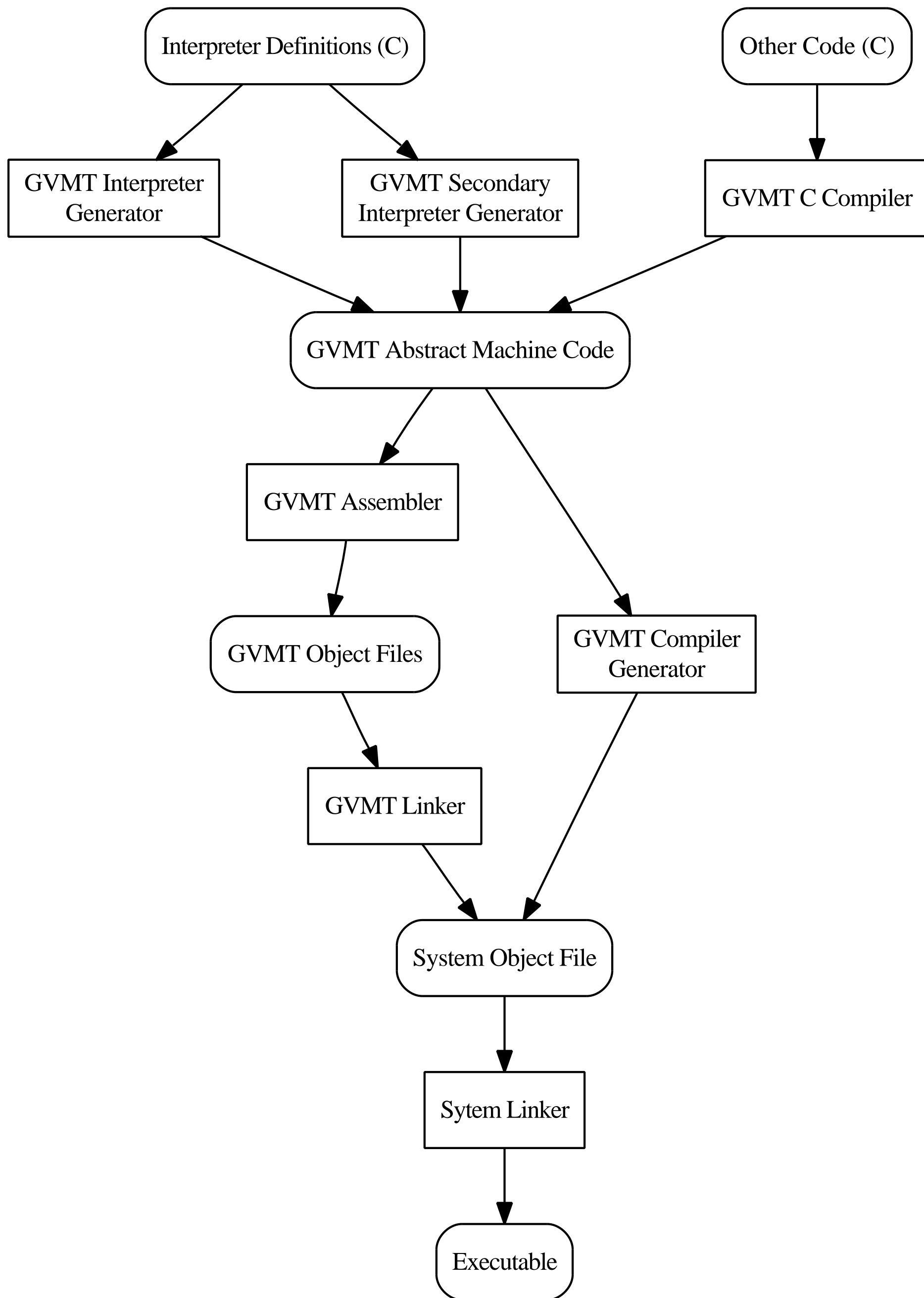


Figure 4.1: The GVMT Tools

based abstract machine that is suitable as a target for a C compiler. It can be translated efficiently into executable code and it supports features necessary for building a VM for dynamic languages.

4.2 The Abstract Machine

The GVMT abstract machine is a stack machine; all arithmetic operations (such as addition) operate on the stack and, like Forth but unlike the JVM, all procedure parameters are moved to and from the stack explicitly. A number of operations for stack manipulation are also provided, in order to assist with the often complex procedure calling semantics of languages like Python. It is also designed to be garbage collection safe throughout.

4.2.1 The Abstract Machine Model

The GVMT abstract machine consists of one or more threads of execution and main memory. Each thread consists of three stacks: the data stack, used for evaluating expressions and passing parameters; the control stack, which holds activation records for procedures; and the state stack used to save the abstract machine state. The state stack is used to implement exceptions, closures, or other complex flow control. See Figure 4.2. The GVMT abstract machine is also fully thread safe and provides features to support concurrency in the VM.

The main memory of the GVMT abstract machine contains two distinct regions, a garbage collected heap and user-managed memory. All pointer instructions differentiate between pointers into the garbage-collected heap and pointers into user-managed memory.

Finally, and possibly most importantly, the abstract machine supports interpreters as special objects. As discussed in Section 3.3.2, this allows the GVMT to produce a JIT compiler automatically and ensure that interpreted code and compiled code behave in the same way. An interpreter is defined by a set of named bytecodes, each one of which is defined by its stack effect and the code describing its semantics. An example of a bytecode defined for the GVMT is given in Section 4.3.2.

4.2.2 Stack-based execution model

A stack-based execution model is chosen for two reasons. The first is simply that most modern VMs are stack-based. The second is that it is generally easier to implement source to bytecode compilers for stack-based intermediate forms. In

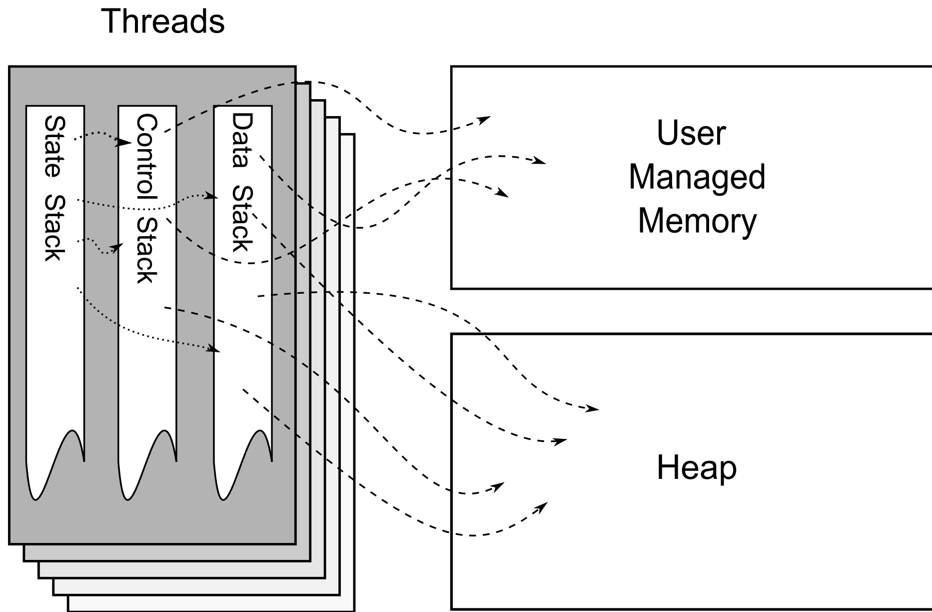


Figure 4.2: The GVMT Abstract Machine Model

terms of performance it does not really matter whether the abstract machine is stack or register based, since a stack-based form is easily interchangeable with a three-address form.

4.2.3 GVMT Abstract Machine Code

As befits an abstract machine code, GVMT abstract machine code (GAMC) has no binary representation; it is purely textual. Instructions are generally of the form `XXX_T` or `XXX_T(N)` where `XXX` is the instruction name, `T` the operand type and `N` is an integer. For example, `ADD_I4` adds two 32-bit integers, whereas `TSTORE_R(N)` stores a `R`eference to the `N`th temporary variable.

The instruction set also has a large number of instructions to provide access to abstract machine features such as the garbage collector and the state stack. The full instruction set is listed in Appendix A. For the full grammar of the GVMT abstract machine code format, including data, see Appendix B

4.2.4 The Stacks

Each thread of execution has three stacks: the data stack, the control stack and the state stack.

Data Stack

All arithmetic operations pop their operands from the data stack and push the result to the data stack. The data stack is kept in thread-local memory, with the top-of-stack determined by the stack pointer, `SP`. `SP` can be accessed and modified directly, allowing the VM implementer a large degree of flexibility. However, it can only be accessed by specific instructions, which gives back-ends some freedom to keep some of the values near the top of the stack in registers; in order to improve performance. Instructions are also provided for block insertions and deletions on the data stack; allowing custom calling conventions and features like C's vararg semantics to be implemented.

Control Stack

The control stack holds local variables for each function activation, as well as any information required by the native ABI[1]. This will usually be integrated with the native stack. The back-end is responsible for ensuring that all references (garbage-collected pointers) stored in the control stack are reachable by the garbage collector.

State Stack

The state stack is used to preserve and restore the machine state. A state object consists of the current point of execution, as well as the current control and data-stack pointers. Instructions are provided to make non-local jumps in execution, restoring the machine state to the state stored in the object on top of the state stack. State objects do not encapsulate the whole machine state; no record is kept of the contents of the heap or of the contents of the data stack, just the depth.

4.2.5 Data Types

The GVMT supports twelve different data types, which are listed in table 4.1; eight integer types (four signed and four unsigned), two floating point types and two pointer types. The GVMT has two different pointer types so that pointers into the garbage-collected heap and pointers into user-managed memory can be correctly differentiated.

Many instructions have a suffix which matches the code of the type. For example, the instruction to perform signed add on two 4-byte integers is `ADD_I4`. The type of data and instruction must generally match, with a few exceptions. Applying a signed operation to an unsigned value implicitly converts it to a signed value,

[1] Application Binary Interface

Kind	Size	Code
Signed Integer	1	I1
Signed Integer	2	I2
Signed Integer	4	I4
Signed Integer	8	I8
Unsigned Integer	1	U1
Unsigned Integer	2	U2
Unsigned Integer	4	U4
Unsigned Integer	8	U8
Floating Point	4	F4
Floating Point	8	F8
(Non-heap) Pointer	4 or 8	P
(Heap) Reference	4 or 8	R

Table 4.1: GVMT Types

and vice versa. The `ADD_P` instruction adds a pointer to an integer, not to another pointer.

The GVMT abstract machine may be either 32 bit (4 bytes), or 64 bit (8 bytes), which determines the size of pointers and references. GVMT abstract machine code is generally not portable from one size to the other, but the types `IPTR` and `UPTR` are provided as aliases for pointer sized integer types.

Data stack items can hold any GVMT data type. When integers smaller than the word size are pushed to the stack, they are extended to word size, retaining their value. Thus signed integers are signed extended and unsigned integers are zero extended. Arithmetic operations on integers compute the full result which is then truncated to the size of the instruction; division rounds towards 0. Floating point operations behave as specified by IEEE 754. The GVMT does not specify byte-order; implementations will match the underlying architecture.

4.2.6 Execution Model

In the following discussion, the term ‘bytecode’ is used below to refer to a virtual-machine instruction and the term ‘instruction’ is used to refer to an abstract-machine instruction. Bytecodes (virtual-machine instructions) are defined by sequences of instructions (abstract-machine instructions).

Execution of a thread starts by creating a new set of stacks for that thread. Initially all stacks are empty. The arguments passed to the `gvmt_start_thread` function are pushed to the data stack, followed by the address of the start function. A `CALL_X` instruction is then executed, where `X` depends on the type specified in the `gvmt_start_thread` function. The `CALL_X` pops the address of the function to be called from the top of the stack and calls it.

Functions

A function in GVMT is defined as a linear sequence of instructions.

Execution of a function procedes as follows: A frame containing all the temporary variables necessary for the function is pushed to the control stack. This frame becomes the current frame for accessing all temporary variables; temporary variables in frames other than the current frame cannot be accessed. The internal layout of this frame is implementation defined. The first instruction in the function is then executed, proceeding to the next instruction and so on. The exceptions to this are flow control instructions, `HOP` and `BRANCH`, which may jump to a designated successor instruction.

Temporary variables are accessed by the `TLOAD_X(n)` and `TSTORE_X(n)` instructions. They have no address and have the same types as data stack elements, with the same restrictions on mixing types.

Interpreters

An interpreter acts externally like a normal function; it can be called like any other. Internally, its behaviour is substantially different from that of a normal function.

The interpreter commences execution, like a normal function, by pushing a frame to the control stack. This frame will have sufficient space to store all the temporary variables of the bytecodes of the interpreter plus any interpreter-scope variables. The interpreter definition specifies the names and types of these variables.

Each activation of an interpreter contains a virtual-machine-level instruction pointer which tells it which bytecode to execute. The start-point of the interpreter is passed in as a parameter and popped from the data-stack on entry.

Execution of bytecodes proceeds in a linear fashion, unless a `JUMP` or `FAR_JUMP` abstract-machine instruction is encountered.

The execution of individual bytecodes proceeds as follows: The abstract-machine instructions that make up that bytecode are executed in the same way as for a normal function. Should the end of the bytecode be reached (as it will be for most bytecodes) then the instruction pointer is updated to point at the next instruction and that instruction is then executed. If a `JUMP` or `FAR_JUMP` abstract-machine instruction is encountered, then the virtual-machine-level instruction pointer is modified, the execution of the current bytecode halts immediately, and the bytecode pointed to the (modified) virtual-machine-level instruction pointer is executed.

Compiled Code

The output of the compiler is a function and can be called like any other. Its behaviour, in GVMT abstract-machine terms[2], is exactly the same as if the interpreter were called with the same input (bytecodes) as passed to the compiler when it generated the compiled function, provided the bytecodes are not modified.

4.3 Front-End Tools

The front-end tools exist to allow the VM developer to program in C, rather than directly in abstract machine code. The tools translate C into abstract machine code. There are three tools; the interpreter generator, GVMTIC, the secondary interpreter generator, GVMTXC and the C compiler, GVMTC. GVMTIC translates interpreter definitions into GAMC. GVMTXC translates secondary interpreter definitions to GAMC, using the output of GVMTIC to ensure that the bytecode format used by primary and secondary interpreters is consistent. The C compiler translates all non-interpreter code and acts likes a standard C compiler with GAMC as its output. The distinction between primary and secondary interpreters is that the primary interpreter defines the bytecode format, whereas the secondary interpreters conform to that format.

The front-end tools accept standard C code[3] with a range of built-in functions to support the various abstract machine features that are not directly supported in C.

4.3.1 The C Compiler

The GVMT C compiler, GVMTC, uses the LCC[35] C compiler with a custom back end. In addition to generating GVMT abstract machine code, GVMTC does simple type analysis to differentiate between heap pointers and other pointers, undoes any unsafe (for garbage collection) optimisations that LCC may have done, and produces error messages for any unsafe use of pointers. Unsafe uses of pointers include the illegal use of pointers to the middle of an object, or attempting to use non-heap pointers as heap pointers (or vice-versa). The GVMT code and documentation refers to heap pointers as *references* and non-heap pointers simply as pointers.

[2]Its real-world behaviour may differ; it should be faster, and it may implement the top of the data-stack differently.

[3]C89 code

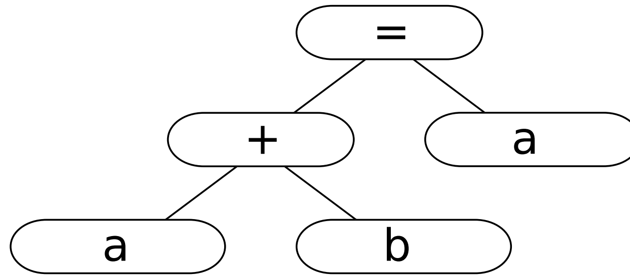


Figure 4.3: Tree for `a += b`

Translating LCC Intermediate Code to GAMC

The intermediate representation used by LCC is a list of trees[30]; each statement in the C source is represented by one or more trees. For example, the C statement `a += b;` is represented by the tree in Figure 4.3. Converting tree representations to stack code can be done by walking the tree bottom-up left-to-right. The tree for `a += b;` can be represented as `a b + a =` in reverse-polish notation. If `a` and `b` are both local variables and four byte integers, then the GAMC code for `a += b;` could be

```
TLOAD_I4(1) TLOAD_I4(2) ADD_I4 TSTORE_I4(1)
```

Looping constructs are converted into explicit branches by the LCC front-end. These are represented in GAMC by the `HOP` instruction for an unconditional jump and `BRANCH_T` or `BRANCH_F` for a conditional jump. All branches must have an explicit `TARGET`.

The following example code is taken from the source code for the HotPy VM. It creates a new string (a heap object) from an array of characters (a non-heap object). The function `gvmt_malloc` creates a new object in the heap.

```
1. R_str string_from_chars(uint16_t* chars, int count) {
2. int i;
3. R_str result = (R_str)gvmt_malloc(sizeof(string_header) + (count << 1));
4. result->ob_type = type_str;
5. result->length = count;
6. for (i = 0; i < count; i++) {
7. result->text[i] = chars[i];
8. }
9. string_hash(result);
10. return result;
11.}
```

This is translated into the following abstract machine code, with `LINE` and `FILE` instructions removed: The numbers at the start of each line correspond to the line numbers above.

```
1.  string_from_chars:
    NAME(0,"chars") TSTORE_P(0) NAME(1,"count") TSTORE_I4(1)
3.  TLOAD_I4(1) 1 LSH_U4 12 ADD_U4 GC_MALLOC NAME(3,"result") TSTORE_R(3)
4.  ADDR(type_str) PLOAD_R TLOAD_R(3) 0 RSTORE_R
5.  TLOAD_I4(1) TLOAD_R(3) 4 RSTORE_U4
6.  0 NAME(2,"i") TSTORE_I4(2) HOP(193) TARGET(194)
7.  TLOAD_I4(2) 1 LSH_I4 TSTORE_I4(5) TLOAD_I4(5) TLOAD_P(0) ADD_P PLOAD_U2
    TLOAD_R(3) 12 TLOAD_I4(5) ADD_I4 RSTORE_U2
6.  TLOAD_I4(2) 1 ADD_I4 TSTORE_I4(2) TARGET(193)
6.  TLOAD_I4(2) TLOAD_I4(1) LT_I4 BRANCH_T(194)
9.  TLOAD_R(3) ADDR(string_hash) CALL_V
10. TLOAD_R(3) RETURN_R ;
```

The translation from the C code works as follows:

Line 1 Line 1 declares two parameters, which are passed on the stack and must be stored into temporary variables with the instructions `T_STORE_P(0)` and `T_STORE_I4(1)`. They are also named for debugging purposes with `NAME(0,"chars")` and `NAME(1,"count")`.

Line 2 Line 2 is just a declaration, so no code is generated.

Line 3 The expression `sizeof(string_header) + (count << 1))` is translated to `TLOAD_I4(1) 1 LSH_U4 12 ADD_U4`. The `gvmt_malloc` function is an intrinsic function, so the call is translated directly to the `GC_MALLOC` instruction.

Line 4 The expression `type_str` is a global variable, so the value is loaded from a fixed address: `ADDR(type_str) PLOAD_R`. Since `result` is a heap reference, a `RSTORE_R` instruction must be used to store the `ob_type` field; internal pointers are forbidden.

Line 5 Line 5 is similar to line 4, except that the `length` field is an integer, so the `RSTORE_U4` instruction is used instead.

Line 6 The `for` statement is three statements in one; an initialisation, a test and an increment. The initialisation, `i = 0` translates to `0 TSTORE_I4(2)` followed by a `HOP` instruction to jump to the end of the loop. The increment and test are emitted after the body of the loop; the increment as `TLOAD_I4(2) 1 ADD_I4 TSTORE_I4(2)` and the test as `TLOAD_I4(2) TLOAD_I4(1) LT_I4 BRANCH_T(194)`.

Line 7 The LCC front-end performs common sub-expression elimination to create the temporary `t5 = i << 1` which is translated as `TLOAD_I4(2) 1 LSH_I4 TSTORE_I4(5)`. The value `chars[i]` becomes `TLOAD_I4(5) TLOAD_P(0) ADD_P PLOAD_U2` which is stored in `result->text[i]` by `TLOAD_R(3) 12 TLOAD_I4(5) ADD_I4 RSTORE_U2`.

Line 9 The `string_hash` function is declared as `void` so is called with a `CALL_V` instruction.

The strict separation between non-heap pointers, designated `P`, and heap references, designated `R`, should be noted. On line 7, loading the character from the

array `chars` uses a `PLOAD_U2` instruction whereas the store into the string `result` uses the `RSTORE_U2` instruction.

4.3.2 The Interpreter Generator

The GVMT Interpreter Generator, GVMTIC, translates an interpreter definition into a GAMC file.

A GVMT interpreter definition consists of two parts: a list of interpreter-scope variables and a list of bytecode definitions. Each bytecode definition consists of an effect declaration and a block of C code. The effect declaration describes the values taken from the stack, operands taken from the instruction stream, and the values pushed back to the stack. The block of C code determines what the bytecode actually does.

The effect declaration of a bytecode takes the form of a Forth-style stack comment: `(inputs -- outputs)`. Inputs may come from the stack, or from the bytecode instruction stream, in which case the name is prefixed with one or more '#' characters. The number of #s indicates the number of bytes to form the value. All inputs and outputs are of the form `type name`.

The following example bytecode definition is taken from the GVMT Scheme implementation (described in Section 4.10). It stores the value currently on top of the stack into the local variable indexed by the next value in the instruction stream.

```
load_local(int #index -- GVMT_Object o) {
    o = frame->values[index];
}
```

The first line gives its name `load_local` and the effect declaration. The effect declaration has one input `int #index` which is a one byte input taken from the instruction stream, and one output `GVMT_Object o` which is a heap object. The second line is the C code which determines what it does; `frame` is an interpreter-scope variable, and is a reference to the Scheme activation frame.

Translation to GAMC

The GVMT interpreter generator, GVMTIC, parses the effect declaration, and delegates the translation of the body to the C compiler, GVMTC. For the example above, GVMTIC translates the `load_local` instruction into the following GVMT abstract machine definition, this time with `LINE` and `FILE` instructions left in:

```
load_local=33:
FILE("interpreter.vmc") LINE(360) #@ NAME(0,"index") TSTORE_I4(0)
LINE(361) TLOAD_I4(0) 2 LSH_I4 TSTORE_I4(3) LADDR(frame)
PLOAD_R 8 TLOAD_I4(3) ADD_I4 RLOAD_R NAME(1,"o") TSTORE_R(1)
```

```
LINE(360) TLOAD_R(1) ;
```

The interpreter generator automatically assigns an opcode to any bytecode definition that does not have one. In this case, line 1, `load_local=33`, shows that GVMTIC has assigned an opcode of 33 to this bytecode.

Inputs taken from the instruction stream are implemented with the `#@` instruction, which takes the next byte from the instruction stream and pushes it to the data stack. The interpreter local variable, `frame`, is not accessed as a temporary, but using the `LADDR` instruction; the expression `frame` is translated to `LADDR(frame) PLOAD_R`. The remaining code is the same as if it were translated by the C compiler, except that there is no trailing `RETURN_X`.

If required, bytecodes can also be defined in a Forth-like style composing instructions out of other instructions and the GAMC instruction set.

4.3.3 The GVMT Secondary-Interpreter Generator

In addition to the main interpreter it is often useful to have additional interpreters that operate on the same instruction set. Examples of these include verifiers, analysis tools and optimisers. The GVMT provides a secondary-interpreter generator, GVMTXC, which can take a partial definition of an interpreter, filling in the missing bytecode definitions with no-ops. The secondary-interpreter generator guarantees that the new interpreter uses exactly the same bytecode format as the main interpreter, producing an error message if any definition conflicts with the primary definition. The secondary-interpreter generator makes the implementation of bytecode dis-assemblers and verifiers simpler and quicker, by ensuring that there is no mismatch in the instruction set, and that no bytecodes have been omitted.

When a secondary interpreter is defined, the secondary-interpreter generator performs two actions. For bytecodes that are specified it verifies that they take the same number of values from the instruction stream as the instruction with the same name in the primary interpreter and that it has the same opcode; the translation to GAMC is performed in exactly the same way as for the primary interpreter. For bytecodes that are unspecified, a bytecode is generated that consumes the same number of values from the instruction stream, but performs no action. A missing `load_local` bytecode from the example above would be translated as `load_local=33: #@ DROP ;` which would consume one byte from the instruction stream and then discard it.

4.3.4 Multiple Interpreters

It is worth pointing out that the GVMT supports multiple primary interpreters in one VM. It is sometimes useful to have more than one primary interpreter in a single VM, for example a VM might require a second interpreter for handling regular expressions. In addition, each primary interpreter can have any number of secondary interpreters.

4.4 Back-End Tools

The back-end tools take the GAMC produced by the front-end tools as input and generate a complete VM. The GAMC is usually generated by the GVMT front end tools, but that is not a necessity. The GVMT back-end tools generate machine code via the native C/C++ compiler, currently GCC, and a JIT-compiler library, currently LLVM.

4.4.1 The GVMT Assembler

The GVMT assembler, GVMTAS, translates GVMT abstract machine code to native object files. It is called an 'assembler' as it converts low-level code to native code, but is rather more complex than most assemblers. GVMTAS uses the native C compiler to generate machine code.

The seemingly redundant translation of GVMT abstract machine code, which was created from C code, back to C code is necessary for two reasons. The first reason is to ensure that garbage collection issues, such as stack and heap layout, are dealt with correctly. The second is to enable the the interpreter generator and compiler generator to share a common low-level bytecode specification.

Translation to C involves stack erasure and handling of the interpreter-level instruction pointer, data stack and frame pointer. It also involves insertion of code to assist garbage collection and to perform non-local jumps. Since the GVMT treats interpreters as special objects, GVMTAS also generates the actual interpreter executable using a similar process. At the real-machine level, generated interpreters are stand-alone functions like any other. The translation process is described in more detail in Section 4.5.

4.4.2 The GVMT Compiler Generator

The GVMT compiler generator, GVMTCC, generates a JIT compiler from an interpreter definition. The input to GVMTCC is an interpreter definition in GAMC

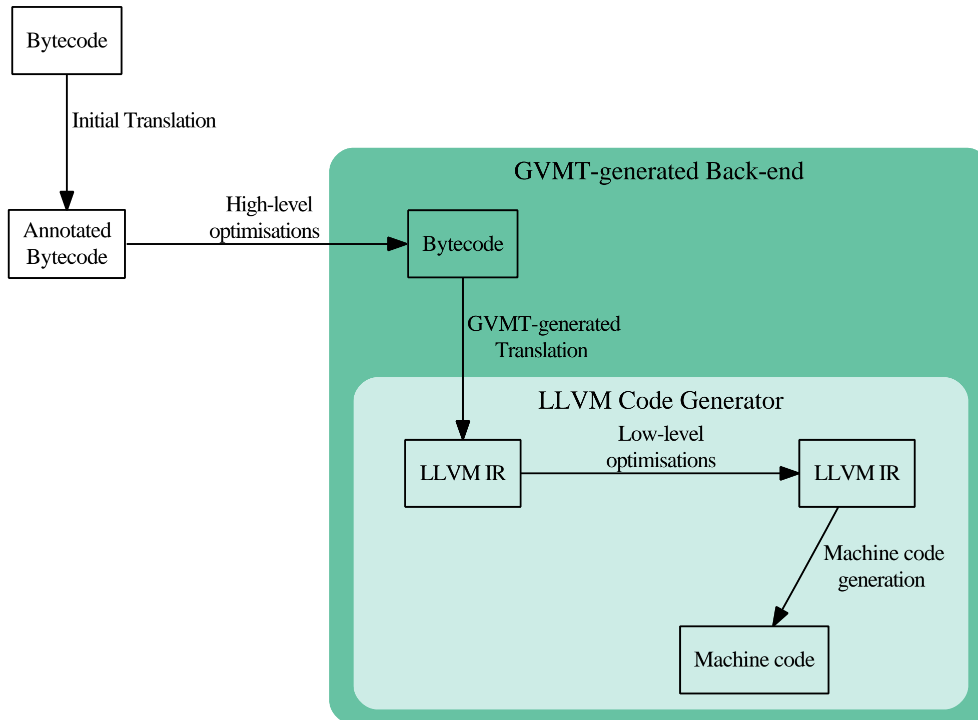


Figure 4.4: The GVMT-built Compiler

form. In other words, the input to GVMTCC is the output from GVMTIC.

Formally, GVMTCC takes an interpreter definition d and produces a compiler c_d, that when given a list of bytecodes b, produces a function f_{db}. Executing f_{db} is equivalent to interpreting the list of bytecodes b with the interpreter i_d generated by GVMTAS from the same interpreter definition d.

$gvmtcc(d) \rightarrow c_d$ Build time
$c_d(b) \rightarrow f_{db}$ Compile time
$f_{db}(_) \equiv i_d(b, _)$ Execution time

Figure 4.5 shows this graphically. In the figure, the generated compiler is both data and a process. It is data as it is the output of GVMTCCİt is also a process which compiles bytecode.

The simplest way to compile a sequence of bytecodes, each of which consists of a list of abstract machine instructions, would be to first concatenate those abstract machine instructions, then translate the resulting (very long) list of abstract machine instructions, instruction by instruction, to native machine code. This would result in a compiler that was doubly inefficient, being both slow and producing slow machine code. The problem with this naïve approach is that the generated code has to do lots of work that could have been be done during compilation, or eliminated all together at build time.

An obvious way to improve this is to use standard compiler techniques to optimise

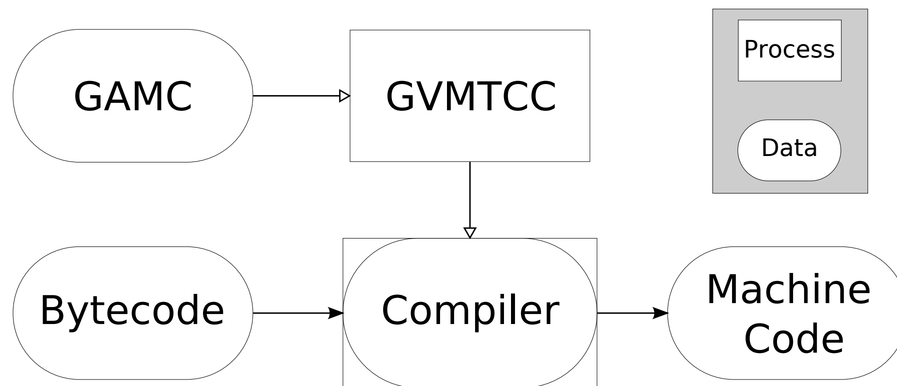


Figure 4.5: The GVMT Compiler Generator

either the abstract machine code or some equivalent, before generating machine code. GVMTCC generated compilers use LLVM[50] to perform machine generation. Rather than generate code to convert individual GAMC instructions to LLVM form, GVMTCC uses partial evaluation techniques to generate code that can generate LLVM intermediate representation directly, bytecode at a time, without passing through the GAMC representation. LLVM can then do further analysis at runtime before generating machine code.

An important step when translating from stack-based code to three-address form is stack erasure. GVMTCC does *intra*-bytecode stack erasure at build time and the GVMTCC generated compiler does *inter*-bytecode stack erasure at compile time.

Although GVMTCC is currently reliant on LLVM[50] to do its final machine-code generation, other options such as using libJIT or a custom-back end are possible.

4.5 Translating GVMT Abstract Machine Code to Real Machine Code

In order to create VMs that perform well, the abstract machine must be mapped efficiently onto the hardware. Mapping the GVMT abstract machine to a real machine primarily involves converting the abstract machine code into real machine code via either the native C compiler or a JIT-compiler library, currently LLVM.

4.5.1 Stack Erasure

The first stage in transforming the GVMT abstract machine code into real machine code is to eliminate as much stack traffic as possible by converting the code to three address form. For example, the sequence:

```
TLOAD_I4(0) TLOAD_I4(1) ADD_I4 TSTORE_I4(2)
```

can be converted to the-three address form:

```
t2 = t0 + t1;
```

Not all stack traffic can be eliminated; the stack is used for parameter passing and the memory in the stack can be accessed directly by the VM developer. Therefore an actual stack must exist. For example, the sequence:

```
TLOAD_I4(0) MUL_I4 TSTORE_I4(2)
```

cannot be converted directly to three address form, as there are insufficient operands available for the `MUL_I4` instruction. An explicit pop from the memory stack must be inserted. The resulting code is:

```
s0 = stack_pop();
t2 = s0 + t0;
```

Similarly, stack pushes are sometimes required. The sequence:

```
TLOAD_I4(0) TLOAD_I4(1) ADD_I4
```

must push the final value to the stack. The resulting code is:

```
s0 = t0 + t1;
stack_push(s0);
```

The stack is implemented simply with a dedicated region of memory and a stack pointer. There is one stack pointer, SP, per thread, and it is used frequently, so ideally it should be kept in a register.

Translating to C Code

With the exception of the JIT compiler output, all GVMT abstract machine code is translated to machine code via C. For efficiency reasons some of the generated code may be tailored to the specific architecture and compiler, but it is generally portable.

Translation of most instructions that operate on the stack is preceded by stack erasure to produce three address code. This three address code can then be emitted as a series of C statements. Almost all arithmetic and logical operators map directly to the C equivalent, but some care needs to taken with signed and unsigned values. For example, the `RSH_I4` instruction performs a signed arithmetic right shift, but the C standard does not state whether the operator `>>` is arithmetic or logical for signed values. Therefore `RSH_I4` cannot be directly translated as `x >> n`. For those architectures which perform logical shifts the following expression is used:

```
((-(x<0))&(~(-1>>n)))|(x>>n)
```

The flow control instructions, `HOP` and `BRANCH`, can be encoded as simple `goto` statements. Translation of other instructions depends on the memory management subsystem, discussed in the next section, and on the implementation of the stacks.

As an example consider the definition of the `load_local` bytecode from Section 4.3.2.

```
load_local(int #index — GVMT_Object o) {
    o = frame->values[index];
}
```

which translates into the GVMT abstract machine code:

```
load_local=33:
FILE("interpreter.vmc") LINE(360) #@ NAME(0,"index") TSTORE_I4(0)
LINE(361) TLOAD_I4(0) 2 LSH_I4 TSTORE_I4(3) LADDR(frame)
PLOAD_R 8 TLOAD_I4(3) ADD_I4 RLOAD_R NAME(1,"o") TSTORE_R(1)
LINE(360) TLOAD_R(1) ;
```

Since `load_local` is a bytecode, rather than a function, it will be wrapped in a switch statement as part of the interpreter dispatch loop. Each bytecode is a `case` statement, plus the declaration of any variables required.

`load_local` is translated to C as follows (the following code is the actual output from the assembler, GVMTAS):

```
1. case _gvmt_opcode_interpreter_load_local:
/* Deltas 1 0 1 */ {
2. GVMT_Object gvmt_r137; {
3. int32_t index; GVMT_Object o; int32_t gvmt_t3; /* Mem temps [] */
4. #line 360 "interpreter.vmc"
5. index = _gvmt_ip[1];
6. #line 361 "interpreter.vmc"
7. gvmt_t3 = (index<<2); \
8. o = (((GVMT_memory*)(((char*)(gvmt_frame.frame))+(8+gvmt_t3)))->R);
9. #line 360 "interpreter.vmc"
10. gvmt_r137 = o; }
11. _gvmt_ip += 2; gvmt_sp[-1].o = gvmt_r137; gvmt_sp -= 1; } break;
```

This is explained, line by line, as follows:

Line 1 The `case` statement for dispatching. The comment `/* Deltas 1 0 1 */` describes the number of instruction bytes consumed, the number of stack values consumed and the number of stack values produced, respectively.
Line 2 Declares a variable used as top of stack.
Line 3 Declares the explicitly named temporary variables.
Line 4 Declares the line number and file for the debugging information.
Line 5 The translation of `#@ TSTORE_I4(0)`. The variable `_gvmt_ip` is the instruction pointer.

Line 6 As line 4.

Line 7 The translation of `TLOAD_I4(0) 2 LSH_I4 TSTORE_I4(3)`

Line 8 The sub-expression `LADDR(frame) PLOAD_R` translates to `gvmt_frame.frame`. The variable `gvmt_frame` is a C struct holding the interpreter-scope variables.

Line 9 As line 4.

Line 10 The translation of `TLOAD_R(1)`. GVMT tries to maintain the top values of the stack in registers, rather than in memory. The variable `gvmt_r137` is used to hold the top of stack value.

Line 11 Adjusts the instruction pointer, `_gvmt_ip += 2`, saves `gvmt_r137` to the memory stack, `gvmt_sp[-1].o = gvmt_r137`, and adjusts the stack pointer, `gvmt_sp -= 1`.

4.5.2 Memory

GVMT memory is divided into two parts: a garbage-collected part, or heap, and a user-managed part. For the user-managed part of the memory, the abstract-machine model corresponds directly to the memory model of C and maps directly to the hardware.

Implementing the garbage collected part of memory requires a garbage collector to be implemented, and the interface between the rest of the abstract machine and the heap to be defined. The GVMT garbage collector is discussed in Section 4.6. The interface between the generated code and the heap consists of four parts: allocation, GC safe-points, barriers and identification of pointers into the heap. For bump-pointer allocators, discussed in Section 2.3.1, the fast allocation path is inlined into the code and the slower fallback implemented with a call to the garbage collector. Both GVMTAS and GVMTCC perform memory-management improvements, such as removing redundant stores during object initialisation, discussed in Section 4.6.5, and inlining of write barriers, discussed in Section 4.6.2.

GC safe-points are implemented as a test of a global variable to see if garbage collection is pending. If it is, then a call to the garbage-collector is made. All generated code conforms to the same convention for layout of frames in the control stack, in order to ensure that the garbage collector can correctly identify all pointers into the heap.

4.5.3 The Control Stack

The control stack consists of values that are not garbage-collected (integers, floating-point values and user-managed pointers) and references to garbage-collected values which need to be scanned during garbage collection. The control stack is thus implemented as a singly linked list of blocks of references interspersed with non-references and whatever bookkeeping values the native ABI re-

quires. The first node in the linked list is the current frame, and is pointed to by a thread-local frame-pointer.

This approach, and how to implement it in automatically-generated C code, is described in more detail by Henderson [36]. When implemented naïvely, this can result in excessive memory traffic. The number of explicit memory accesses required can be reduced by using liveness analysis; if a reference is not live across a GC safe point, it can be ignored. Jung et al. [48] describe the use of this technique in a Java-to-C compiler.

An alternative approach would be to record the offset information for each reference in the control-stack frame in a table. Although this would probably be faster, it is impossible in portable C or with LLVM. The cost of maintaining the linked list does not seem to be a problem.

4.5.4 Handling the Stack Pointer and Frame Pointer

Since the GVMT frame-pointer can be synthesised cheaply anywhere that the C struct implementing the topmost frame is in scope, the only time that the GVMT frame-pointer needs to be made explicit is when calling a procedure, so that the newly created frame can be linked into the control stack. The frame pointer, FP, can be synthesised by the C code:

```
FP = &gvmt_frame;
```

where gvmt_frame is the C struct for the control-stack frame. This translates into a single machine instruction:

```
FP = %fp_register + fixed_offset
```

The stack pointer, SP, is required throughout the program and may be modified across calls. However, its exact management can be left to the C compiler or LLVM, provided that its value is made explicit at both call and return sites. This suggests the following strategy for calls and returns:

At call sites: pass FP and SP in registers. Pass the top-of-stack value(s) in registers if the architecture allows it. The x86 architecture only allows two parameters to be passed in registers, so the GVMT stack must be pushed to memory at call sites.

At return sites: If the machine ABI supports two return registers then return the function result in one and SP in the other. If the machine ABI supports only one return register, like the x86, then return the SP in a register and the function result on the stack.

4.5.5 The State Stack and Execution Control

In order to save and resume the execution state of the abstract machine it is necessary to save not only the data stack pointer SP, but also the state of the control stack and the current point of execution. The current point of execution includes both the interpreter's instruction pointer and the hardware instruction pointer.

This requires saving the state of the real machine, using something akin to C's `setjump-longjump` mechanism. For the x86 implementation, a custom function for saving state (`setjump`) and restoring state (`longjump`) were written in assembler. Although this is not portable, it is fewer than 20 lines of assembler and should be easy to adapt to other architectures.

4.6 Memory Management in the GVMT

The GVMT heap organisation is designed to support garbage collection without dictating the garbage collection algorithm. Whilst it is impossible to predict all requirements, generalisations can be made. As discussed in Section 2.3.7, the following requirements are postulated as likely for most, if not all, dynamic languages:

- Allocation is frequent, with many objects dying young (the weak generational hypothesis holds).
- The size of heap may vary widely at runtime as the same VM may be used for running both small scripts and sizeable applications.
- In order to allow the VM to be embedded or to use pre-existing libraries, objects may need to be 'pinned', that is, it may be required of the garbage collector that it does not move certain objects.

It should also be noted that the GVMT heap organisation exists to help create GVMT memory managers. The GVMT abstract machine model is completely independent of the heap organisation. An entirely new heap organisation could be used without affecting the other components of the toolkit, with the exception of the linker.

4.6.1 The Three Levels of Memory Hierarchy

The GVMT heap is organised in a hierarchical fashion. This organisation allows easy resizing of the different regions of the heap, and allows chunks of memory to be transferred between different logical areas without physically moving them.

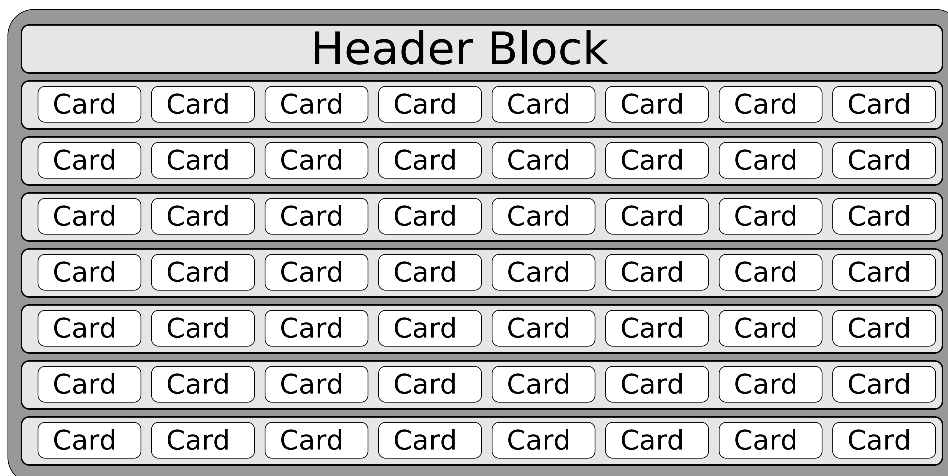


Figure 4.6: A Memory Zone Consisting of Eight Blocks

There are three components in the GVMT memory hierarchy: *zones*, *blocks* and *cards*. Zones are composed of blocks, which are composed of cards.

The GVMT heap organisation extends the BIBOP and page-based organisations discussed in Section 2.3.6. The extra level (Zone) above the page (or Block) is added so that information about a block can be stored outside of the block without requiring a global table. It also allows better separation of garbage collector data structures from the heap objects.

Zones are the units of memory used by GVMT to interact with the operating system. *Blocks* are the chunks of memory that are passed between the various components of the garbage collector. *Cards* are used for finer-grained operations, such as finding inter-generational pointers and for pinning. Figure 4.6 shows a zone of eight blocks, each containing eight cards. The first block is used as a header, rather than containing cards. A real zone would contain more than eight blocks, each containing more than eight cards.

All components are aligned to a fixed power of two. Additionally the size of cards and blocks match their alignment; the size of a zone must be a multiple of its alignment. Although the sizes of cards, blocks and zones can be varied across implementations they must, for performance reasons, be determined at build time.

Addresses and Indices of Memory Chunks

This insistence on power of two alignment allows a number of important garbage collection features to be implemented efficiently on a fragmented heap. The zone, block or card containing any word in memory can be found extremely easily using a single bitwise operation; no memory access is required. Similarly the index of any card within a block or of any block within a zone can be also be calculated in

a couple of instructions, with no memory access.

Consider a chunk of memory with a size and alignment of 2^n bytes, and an arbitrary address a of width W bits. The address of the start of the chunk containing a is the most significant $W-n$ bits of a. The offset of a within that chunk is the least significant n bits. This can be readily extended to finding the index of the chunk of size 2^m containing a within the enclosing chunk of size 2^n provided $m < n$. The index of the smaller chunk is the least significant n bits right shifted by m, evaluated in C as `(a & K) >> m` where `K = (1<<n)-1`.

Zones

All memory is acquired from the operating system as zones. Zones are the only memory entity whose size may differ from its alignment. Zones whose size is larger than their alignment are required for handling very large objects.

The first one or two blocks of a zone are used as header blocks. These are not usable for memory allocation as they provide space for the card-marking table, pinning bitmap and, if required, for object-marking bitmaps.

Blocks

Blocks are the most important level in the hierarchy. They are the chunks of memory handed to thread-local allocators by the global allocator, and can serve as the larger region for a mark-region collector.

Blocks are the units of memory that can be transferred between logical spaces[4]; each block belongs to exactly one space. All blocks, except header blocks, are composed wholly of cards, without any additional space. Since they can be virtually 'moved' without being physically moved, they are also useful for supporting pinning in a moving collector. Since pinned objects cannot be moved, when 'copying' a pinned object, the block containing the object is 'virtually copied' by transferring ownership of the block to the target space. The space to which a block belongs is unrelated to the zone in which it is physically located.

Cards

Card are the lowest level of the hierarchy. Cards are used for inter-generational pointer recording[71] and for mark-region collectors[17]. Cards are fairly unimportant compared with blocks and zones; only their size is of interest as this de-

[4]The term 'space' is generally used in garbage-collection literature to refer to an area which is logically rather than physically distinct.

Figure 4.7: Address word (most significant bit to the left)

termines the amount of space required in the header blocks for internal data structures.

Sizes and Alignments

All alignments, whether for cards, blocks or zones, are powers of two. This means that the layout of a zone can be described by three integers: *Log*$_2$*CardSize* (LCS), *Log*$_2$*BlockSize* (LBS) and *Log*$_2$*ZoneAlignment* (LZS). Zones larger than their alignment are only used for very large objects and are not divided into blocks, but do contain a header. The size of most zones is equal to the zone alignment.

This means that for the GVMT heap layout, the address of the Zone containing address *a* is `a & (-(1<<LZS))` and the index of a Line within a Block is `(a & ((1<<LBS)-1)) >> LCS`. This is illustrated in Figure 4.7.

For example, suppose the chunk sizes were chosen so that LCS = 8, LBS = 16 and LZS = 24 for a 32 bit address space. For an address 0xA1B2C3D4 the Zone address would be 0xA1000000, the Block address would be 0xA1B20000 and the Card address would be 0xA1B2C300. The index of the Block within the Zone would be 0xB2, the index of the Card within the Block would be 0xC3, and the index of the Card within the Zone would be 0xB2C3.

Header Blocks

The number of header blocks depends on the size of the data structures required by the garbage-collection algorithm used, so the following is an example only. The garbage collector used for the HotPy VM is a generational collector, with an Immix[17] mature-space collector and support for pinning. As a generational collector, a card-marking table of one byte per card is required. As a marking collector, the Immix collector requires a bitmap of one bit per word, as well as one byte per card and one word per block for internal book-keeping. Finally, pinning requires one bit per card. The card-marking table should start at the beginning of the zone; see Section 4.6.2 for the reasons. The alignment of the other data structures is less performance critical and they are laid out to minimise space usage.

Choosing the Sizes

As long as there is sufficient room for the necessary data-structures, the card, block and zone sizes should be chosen to maximise performance.

Cards can serve both as lines for a mark-region collector and as the cards in a card-marking collector. In the case of card-marking, a 128 byte card size seems to give the best trade-off between accuracy and space overhead. Empirical evidence suggests that 128 bytes is also the best size for lines in the Immix mark-region collector. The block size should be a multiple of the virtual memory page size, but this is easy to achieve as virtual-memory pages are usually smaller than the ideal block size. Since the card-marking table is heavily used in a generational system, it may help performance if its size is a multiple of the virtual-memory page size. The size of the card-marking table is the number of cards per zone, (Zone Size/Card Size). For example, pages are 4096 bytes in the x86 architecture, so $(ZoneSize/CardSize) \geq 4096 \Rightarrow LZS - LCS \geq 12$.

The Current GVMT Zone Implementation

Currently in the GVMT, cards are 128 (2^7) bytes and blocks are 32k (2^{15}) bytes. Zone alignment is 512k (2^{19}) bytes. Zone sizes can be any integral multiple of the alignment. These values can be readily changed by rebuilding the GVMT. For the generational collector with pinning about 18 kbytes per zone (1.8%) are wasted due to the alignment requirements.

Objects Larger than a Zone

Objects larger than the zone alignment need special handling. In order to accommodate one of these objects, a zone whose size is larger than its alignment is required. This super-sized zone will still have a card-marking table, but no pinning map is required. Additionally, since objects span many blocks, allocation is not done via blocks, so no per-block data is required. Therefore, the object can start immediately after the card-marking table.

The fact that an object can be larger than the zone alignment has implications for card-marking. If the zone containing the card-marking table were calculated from the address being written to, the byte to be marked could be in the middle of an object. Therefore, the zone containing the card-mark must be the zone containing the start of the object being written into, regardless of whether the card-index is determined by the object or the slot written to. There are two corollaries of this: the card-marking table is no larger for a super-sized zone than for a normal zone, regardless of the object size, and for an object spanning N zones, each card-mark can refer to N different cards.

4.6.2 Write-Barriers

As discussed above, the GVMT memory layout includes card-marking tables. Card marking is fast as the zone address can be calculated with a single `and` instruction and the card index computed with two operations, `and` and `shift`. Thus the calculation of the mark address is only four instructions and, unlike a card-marking scheme with a single global table, no register or global variable is required to hold the table address.

Cards can either be marked according to the object written to, or the slot written to. Since the size of an object may exceed the alignment of a zone, the card-marking table is always determined by the object address. The card index may, however, be determined by the object or the field written to. Whether cards are marked by object or by field depends on the garbage collector in use; see Algorithms 4.1 and 4.2.

In the following algorithms the & operator is the bitwise-and operator, and $\overset{u}{\gg}$ is the unsigned right-shift operator. The card-marking table is aligned with the start of the zone.

Algorithm 4.1 Card-marking by object address

$zone \leftarrow object\&(-2^{LZS})$
$card_index \leftarrow (object\&(2^{LZS}-1)) \overset{u}{\gg} LCS$
$zone[card_index] = 1$

Marking by object can be implemented in five instructions for the x86 architecture (object address in register `%edx`):

```
movl %edx, %eax
andl $1048575, %edx
andl $-1048576, %eax
shrl $7, %edx
movb $1, (%eax,%edx)
```

Algorithm 4.2 Card-marking by slot address

$zone \leftarrow object\&(-2^{LZS})$
$slot = object + offset$
$card_index \leftarrow (slot\&(2^{LZS}-1)) \overset{u}{\gg} LCS$
$zone[card_index] = 1$

Marking by field can be implemented in six instructions for the x86 architecture (object address in register `%edx`, offset in register `%ecx`):

```
movl %edx, %eax
```

```
addl %ecx, %edx
andl $1048575, %edx
andl $-1048576, %eax
shrl $7, %edx
movb $1, (%eax,%edx)
```

Algorithm 4.3 Conventional Card-marking

$card_index \leftarrow object \overset{u}{\gg} LCS$
$card_mark_table[card_index] = 1$

By way of comparison the write barrier used in the Self VM[22] for card-marking is listed in Algorithm 4.3 Although it might seem that the overhead for using a fragmented heap is excessive, taking five or size instructions rather than the standard two or three, the standard method needs to find the address of the card-mark table. This either requires a dedicated register (which is not practical for the x86) or it must be read from memory:

```
movl card_mark_table, %eax
shrl $7, %edx
movb $1, (%eax,%edx)
```

The memory-read instruction is likely to cost more than three ALU instructions, meaning that the GVMT write-barrier may be faster than the standard sequence. Since the overhead of card-marking is usually in the order of 1% of optimised compiled code[16], it does not really matter whether the GVMT write-barrier is a bit faster, or a bit slower.

4.6.3 Allocation

The motivation for the hierarchical memory organisation is to allow copying collection to co-exist with object-pinning, and the main reason that copying collection is desirable is that it allows fast object allocation.

Bump-Pointer Allocation

The fastest way to allocate new objects is simply to increment (or decrement) a pointer. Obviously some sort of check is required to ensure that the pointer does not exceed the limits of the available space. Algorithm 4.4 shows the naïve algorithm; $free$ is the pointer to the beginning of free memory.

In order to support concurrent allocation, the free-pointer and the limit-pointer must be thread-local. This means either that they are relatively expensive to read and write or that they require dedicated registers.

Algorithm 4.4 Naïve Bump-pointer Allocation

```
if size + free < limit_pointer then
    result = free
    free = free + size
else
    result = call_allocator(size)
end if
```

The powers-of-two nature of the GVMT heap architecture provides a way of dispensing with the limit-pointer. Memory is handed to the per-thread allocators in blocks of size 2^{LBS}. This means that $limit_pointer = roundup(free, 2^{\mathrm{LBS}})$.

Since $roundup(x, 2^y) = x + ((-x)\&(2^y - 1))$, the limit test can be rewritten as $free + size < free + ((-free)\&(2^{\mathrm{LBS}} - 1))$. This in turn simplifies to $size < ((-free)\&(2^{\mathrm{LBS}} - 1))$. The improved allocation code is shown in Algorithm 4.5.

Algorithm 4.5 Improved Bump-pointer Allocation

```
if size <ᵘ -free&(2^LZS - 1) then
    result = free
    free = free + size
else
    result = call_allocator(size)
end if
```

4.6.4 The GVMT Generational Pinning Collector

As discussed in Section 2.3.7, it is useful for a dynamic language garbage collector to be generational and to support pinning. Although the GVMT supports a number of collectors, the most advanced is the default collector, the Generational-Pinning Collector.

The GVMT generational-pinning collector is designed to provide fast allocation and fast collection combined with the ability to pin objects. The collector is a generational collector, with two generations: a copying nursery and an Immix mature space. It also contains a pinned space, but this is not a separate generation and is collected at the same time as the nursery. The Immix algorithm supports pinning and needs no modification for pinning mature objects.

Pinning of nursery objects is done as follows. When an object is pinned, the object and its enclosing card(s) are marked as pinned. If the enclosing block is not already marked as pinned it is transferred from the nursery to the pinned space. During the next minor collection, blocks in the pinned space are scanned and marked, rather than copied. All blocks in the pinned space are then transferred to the mature space, thus 'virtually copying' the pinned object to the mature space.

After subsequent major collections, any block with no pinned cards remaining is unmarked as pinned and can be used normally.

Overall, the hierarchical block approach gives increased flexibility in the implementation of garbage-collection algorithms, at little or no cost.

4.6.5 Optimising Memory Allocation in the GVMT

Although there is a wealth of publications on garbage collection, the mechanics of allocation are barely mentioned. Memory allocation for a toolkit is more complex than for a single VM and merits some discussion.

When a piece of memory is allocated by the allocator, it must be in a safe state for scanning or garbage collection. This means that it must contain only valid references. Therefore the allocator must ensure that the allocated memory contains only valid data before it is returned to the user program.

For statically-typed languages that ensure that all fields of an object are initialised, there is no need for the allocator to zero the memory, but for a toolkit, which knows very little about the VM, the memory must be made safe. This leads to a number of inefficiencies. Firstly most of the fields of a newly allocated object will be initialised anyway, resulting in redundant code, but worse still, all those initialisations are writes into an object, so they will incur a write-barrier penalty despite the fact that no write barriers are required for newly allocated objects.

The GVMT performs some analysis to remove most of this redundant work. Allocation is split into two: the allocation, and zeroing the memory. The `GC_MALLOC` instruction is split into code to do the allocation, and a `__ZERO_MEMORY` instruction. Subsequent analysis conservatively determines which instructions overwrite which fields in the object. The `__ZERO_MEMORY` instruction is then removed and replaced with a minimal sequences of writes, to zero any field not explicitly initialised. All initialising writes are replaced with equivalents that do not contain a write barrier. The current implementation is quite conservative, so a special intrinsic function, `gvmt_fully_initialised()`, is provided for the VM developer to inform the GVMT that an object has been fully initialised.

4.7 Locks

Although the GVMT is designed to support concurrency and is targeted at dynamic languages, many dynamic languages were not designed with concurrency in mind. The two most popular dynamic languages, Python and Ruby, have evolved in a single-threaded environment, and have features that are awkward to support in a multi-threaded environment. For example, Python list operations,

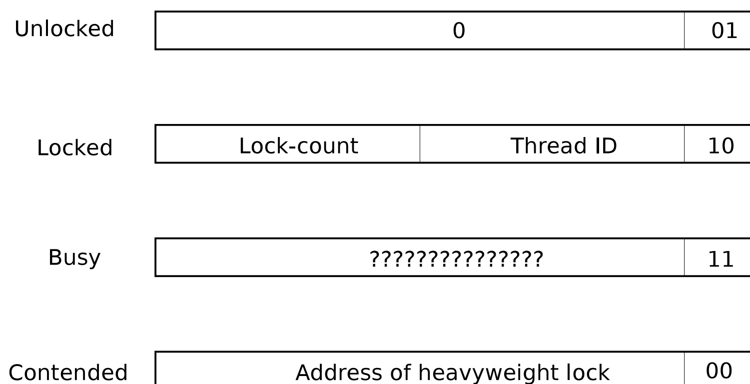


Figure 4.8: Lock representations

such as appending to a list, are implicitly atomic (uninterruptable). Python programmers are likely to be surprised by non-atomic behaviour from such operations, so locking is required for many common operations.

In order to support this high level of synchronisation the GVMT provides a fast, lightweight mutex[5] for the common case where locking operations are unlikely to be contended. For locks that are likely to be contended, the operating system mutex may offer better performance.

The GVMT lock is based on mutexes designed for the JVM, which also requires fast, lightweight mutexes. The GVMT lock is similar in design to 'thin-locks'[7] and 'meta-locks'[2], both developed for the JVM. Unlike the JVM case, the lock is not embedded into the object header (in the GVMT there is no object header), nor is there a requirement, peculiar to Java, that all objects can be used as mutexes. Consequently, a GVMT lock takes a full word of memory. A word is assumed to be 32 bits for the remainder of this discussion, although 64 bit machines would use 64 bit locks.

The word is broken into two parts: the most significant 30 bits, and the least significant two bits. The least significant bits represent four states: unlocked, locked, contended and busy. See Figure 4.8. In the unlocked state the least significant bits are 01 and the other 30 bits are all 0. In the locked state the least significant bits are 10 and the other bits hold the thread id and the lock count. In the busy state the least significant bits are 11 and the other 30 bits are in transition. Finally, in the contended state the full word is a pointer to a heavyweight lock, so the least significant bits are 00.

To lock an unlocked GVMT lock, a single compare-and-swap operation is required, swapping the unlocked value with the thread-specific locked value. Unlocking is equally fast, simply doing the swap in reverse. There are four other cases for locking: recursive locking (relocking a lock already locked by the same thread) which simply increments the lock-count atomically; contended locking on

[5] mutual exclusion lock

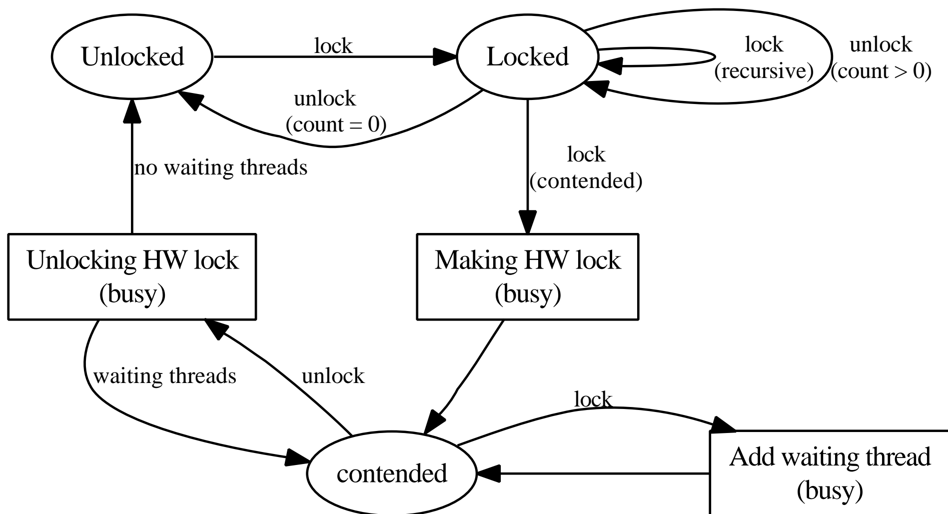


Figure 4.9: Lock states

a previously uncontended lock; contended locking on an already contended lock; and locking a busy lock. Locking a busy lock involves spinning until the lock is no longer busy and then locking.

Acquiring a contended lock is a multi-stage process. A contended lock is a pointer to a heavyweight lock, composed of an operating-system mutex and condition variable. Locking the heavyweight lock is done by waiting on the condition variable before attempting to lock the operating-system mutex. In order to be able to use the heavyweight lock, the GVMT lock must transition from the locked state to the contended state. This is done via the busy state. When a thread wishes to lock a GVMT lock which is locked, but not yet contended, it must atomically change the state to busy, then allocate the heavyweight lock before atomically transitioning to the contended state.

In order to prevent heavyweight locks from being freed while other threads are waiting on them, modifications to the count of waiting threads can be made only with the lock state as busy. See Figure 4.9 for the state transition diagram; oval nodes are stable states, rectangular nodes are transition (busy) states.

4.8 Concurrency and Garbage Collection

Since the GVMT supports concurrency and offers garbage collection, the garbage collector must work correctly in a concurrent environment. The memory management cycle can be viewed as having three parts: allocation, synchronisation and collection.

4.8.1 Concurrent Allocation

The current GVMT garbage collector is a generational collector (see Section 2.3.4). Assuming that the vast majority of allocations are of small objects, only allocation from the nursery need be concurrent. Larger objects are allocated by a single allocator protected with a global lock.

Each thread of execution has its own allocator. Each allocator can then allocate objects without any synchronisation being required. When an allocator runs out of memory, it acquires a new block from the global pool.

4.8.2 Synchronisation

Since the GVMT garbage collector is a stop-the-world collector, all mutator threads must be stopped before the garbage collection can start. When an allocator fails to acquire a new block from the global pool, it signals that garbage collection is to occur. It does this by setting a global flag, `gvmt_collector_waiting`, reducing the running-thread count by one, and waiting for completion of garbage collection. When the running-thread count reaches zero, the collector may start. All running-thread count modifications are performed atomically. Whenever a thread encounters a GC safe point (a `GC_SAFE` instruction) it tests to see if the `gvmt_collector_waiting` flag has been set, and if it has, it decrements the running-thread count and waits for the garbage collector to complete.

There is a problem with this simple running-thread count scheme, as it prevents garbage collection from happening if any thread is performing some slow operation, such as waiting for an internet packet. Consequently, it must be possible to perform garbage collection when threads are executing native code. When native code is entered, the running-thread count must be decremented. It is then incremented when the thread returns from the foreign call. However, when garbage collection is happening, threads must be halted should they return from native code.

To prevent a thread restarting during garbage collection, the following convention is observed: in order to modify the running-thread count *from* zero, a dedicated mutex must be acquired. The garbage collector holds this mutex when it is collecting, preventing any thread from restarting. Finally, to prevent expensive locking and unlocking in single-threaded code, a 'dummy' thread is created to increase the running-thread count by one. The first thread to request garbage collection 'stops' this thread, which is restarted by the garbage collector immediately after completion of garbage collection.

4.8.3 Concurrency within the Collector

Currently the collector is single-threaded. Concurrency could be supported in two ways, either by running some of the collection concurrently with the program, or by using several concurrent threads to do the collection. There are numerous approaches to concurrent garbage collection, many of them deriving from the Mostly-Concurrent algorithm of Printezis and Detlefs[61]. Marlow et al.[53] describe the techniques used to implement a parallel garbage collector in the Glasgow Haskell Compiler.

4.9 Comparison of PyPy and GVMT

PyPy and the GVMT have a common purpose, simplifying the creation of a VM for dynamic languages. Both PyPy and the GVMT provide garbage collection and can automatically generate a JIT compiler, but they differ in choice of input languages, level of automation, complexity, and design philosophy.

The design of PyPy is based on the premise that implementing a VM in a very high level language, namely Python, will simplify the implementation, with attendant benefits in flexibility and maintainability. This is done by pushing as much of the complexity as possible into the tools, in order to hide it from the developer. PyPy aims to minimise the cost of implementing the VM, at the cost of increasing the complexity of the tool set. The design of the GVMT considers the effort required to implement both the toolkit and the VM. The total cost, both of the VM and the toolkit, should be minimised. The GVMT design assumes that the toolkit will be used for relatively few VMs, whereas the PyPy design assumes that the tools will be used for many different VM implementations.

The input language to PyPy is RPython, a dialect of Python. GVMT takes C as its input language, enhanced with a number of built-in functions to access important abstract-machine features such as the stack and garbage collector. Although RPython is undoubtedly more expressive than C, even C enhanced with garbage collection and exception handling, it may not be that much more expressive in a VM implementation. A VM is an inherently low-level system.

The choice of input language is more than a cosmetic difference as it affects the complexity of the tool set to a large degree. Converting C to abstract machine-code is a straightforward task involving a modified portable C compiler. Converting from RPython to a low-level form involves a complex mixture of partial evaluation and whole-program type inference[66].

PyPy and GVMT also differ in their approach to JIT-compiler generation. Both tool sets are capable of generating a JIT compiler from an interpreter specification. PyPy produces a trace-based compiler that performs several optimisations tailored to dynamic languages, such as specialisation and escape analysis. The GVMT

compiler performs conventional compiler optimisations only. The PyPy generated compiler is undoubtedly the more powerful of the two in the context of dynamic languages. However, by optimising at the bytecode level, and using language-specific optimisations that are unavailable to an automatically-generated compiler, a more powerful optimisation system can be built with the GVMT. Chapter 5 describes how this can be done and Chapter 6 shows that the performance of the two approaches is broadly comparable.

The GVMT has two advantages over PyPy. It supports multiple threads of execution and has a better method of supporting integration with exist C libraries.

Support for multiple threads of execution was designed into the GVMT. It provides lightweight locks, which can be embedded in heap objects. Its memory allocator is multi-threaded, and although the collector is single-threaded, it is thread safe. Although PyPy has a global interpreter lock, this is not an inherent limitation in the design of PyPy, but of its current implementation.

The GVMT supports integration with existing C libraries in two ways. The first is almost incidental; thanks to the comparatively low level of the GVMT abstract machine, it maps to the C execution model quite cleanly. The second is deliberate; the garbage collector supports pinning. This allows heap allocated objects to be passed safely to C library code, which can execute concurrently with the garbage collector.

4.10 The GVMT Scheme Example Implementation

The GVMT distribution includes an example virtual machine, GVMT-Scheme. GVMT-Scheme was designed and implemented with the following goals:

- Provide a clear implementation that illustrates how to use the GVMT.
- Be implementable in about two weeks[6]
- Implement enough of Scheme to provide a meaningful performance comparison with other Scheme implementations.

GVMT-Scheme does not provide the full Scheme number tower, just integers and floating-point numbers. However, it does have full runtime type checking, which is one of the two main overheads that a Scheme implementation must handle; the other being garbage collection. Since GVMT-scheme is not a full implementation it is unfair to compare its code size to that of other Schemes; GVMT-Scheme is under 4000 lines of code. GVMT-Scheme contains a precise garbage collector and a JIT compiler, both provided by the GVMT.

[6]The implementation actually took just under three weeks.

4.10.1 Implementation Details

All Schemes execute what is called a read-eval-print loop. In GVMT-Scheme, the 'read' part is implemented by parsing the source code to form an Abstract Syntax Tree, translating the AST to bytecode, and performing simple tail-recursion elimination on the bytecode. The 'eval' part of the read-eval-print loop is implemented by executing the bytecodes. The first time a sequence of bytecodes is evaluated, its bytecodes are interpreted. The second time a sequence of bytecodes is evaluated, the bytecodes are optimised and JIT compiled. Subsequent evaluations are performed by executing the generated machine code.

Integers are tagged, but all other data types are boxed. Frames are allocated on the heap, in order to support closures.

Optimisation in GVMT-Scheme is performed in five passes, four bytecode-to-bytecode optimisation passes generated by the GVMT secondary interpreter generator, GVMTXC, followed by JIT compiler generated by GVMT compiler generator. The bytecode-to-bytecode optimisation passes are:

- Determine if it is possible to remove frames for this closure
- Remove frames if possible.
- Jump threading; remove jumps to jumps and jumps to returns.
- Load-store elimination; remove dead stores, convert store-load pairs to copy-store pairs.

The optimisers are implemented in less than 500 lines of code.

4.11 Conclusions

The GVMT is designed around a stack-based abstract machine that provides garbage collection. As described in Chapter 3, the use of an abstract machine allows the separation of the front-end tools from the back-end tools. The front-ends tools of the GVMT, the C compiler (GVMTC), the interpreter generator (GVMTIC), and the secondary interpreter generator (GVMTXC), convert source code to GVMT abstract machine code, in a way that is largely independent of the target architecture. The back-end tools, the assembler (GVMTAS), the compiler generator (GVMTCC) and the GVMT linker, convert the abstract machine code into executable code. The back-end tools were designed and implemented separately from the front-end tools.

The implementation of the abstract machine, that is the mapping of abstract machine to real machine, has been performed reasonably efficiently for the x86 architecture. It has implemented without using any unusual features of the x86

architectures. A new implemention, for a different architecture, should be able to reuse much of the design and some of the code of the x86 implementation.

As described in Section 2.3.7, dynamic languages require rapid allocation of memory, for short-lived objects, and the ability to pin objects in memory, for interfacing with library code. The GVMT memory management system is able to meet both these requirements using a novel heap layout.

4.11.1 Future Work

Toolkits are, almost by definition, never complete. A wide range of tools and features could be added to the GVMT.

One feature that would be useful is a new compiler back-end. The current LLVM-based back end has a large memory footprint and its compilation speed is rather slow for a just-in-time compiler. Although a new compiler back end is unlikely to produce code that runs as fast as that produced by LLVM, it could be expected to produce that code more quickly and use less memory.

Chapter 5

HotPy, A New VM for Python

This Chapter introduces and discusses the HotPy VM for Python. First, the model of execution of the VM is outlined. The design of the VM, particularly its optimisation control is discussed. The optimisation stages are then covered, noting that the optimisers all work as bytecode-to-bytecode translations as advocated in Chapter 3. An extended example of operation is then given. Finally, HotPy is compared to similar work.

5.1 Introduction

The HotPy virtual machine is a VM for Python, built using the GVMT. 'HotPy' is a recursive acronym for HotPy Optimising Tracing Python. HotPy implements the 3.x series of the language, rather than the more widely used 2.x series. The complete source code and some documentation is available from `http://code.google.com/p/hotpy/`.

HotPy is largely a 'proof of concept' for a high-performance, dynamic-language VM which is built using a toolkit. All the features that make Python interesting, and difficult to implement efficiently, are included: iterators, generators, closures and the ability to manipulate almost any object or class at runtime. Although the core VM and objects are implemented, library support is far from complete.

The design of HotPy is driven by the idea, discussed in Chapter 3, that bytecode is a good intermediate representation for optimisation. HotPy is thus designed as a high-performance *interpreter* foremost. HotPy optimises frequently executed parts of the code, as do most high-performance VMs, but continues to interpret the optimised bytecodes until they become sufficiently 'hot' to be worth compiling to machine code. Compilation is performed by a GVMT-built JIT compiler.

The HotPy VM can thus be classified as a tracing-specialising interpreter, with a JIT compiler. This means that all optimisations specific to Python are handled

within the interpreter, leaving the GVMT-built compiler to do low-level optimisations and machine-code generation.

5.2 The HotPy VM Model

The HotPy VM performs many optimisations in order to achieve good performance. So that the optimisations it performs can be understood and analysed, there must be a means to describe the state of the VM.

5.2.1 The High Level Model

The HotPy VM consists of a single, global garbage-collected heap of objects, one or more GVMT-level threads of execution and one or more HotPy threads. Each GVMT-level thread executes one HotPy thread at a time. In the HotPy VM model, each GVMT-level thread consists of a single reference to a HotPy `thread` object and the GVMT-provided data-stack.

The semantics of HotPy can be defined as if were just a bytecode interpreter, without compilation. The bytecode-instruction pointer is managed by the GVMT-generated components, but it is visible to the HotPy VM. Allocation of objects and garbage-collection is managed by the GVMT.

5.2.2 Execution

Threads

Each HotPy thread is described by a single `thread` object. The current state of execution is described by a stack of `frame` objects, implemented as a singly linked list.

Before describing an executing thread, it is easier to describe a suspended thread. Each `frame` has a `return_ip` which points to the next bytecode to be executed when that frame is the current frame. For each `try` statement that has been executed and is still in scope, there exists one `exception_handler` object. These `exception_handler` objects are attached to the relevant frame to form a chain. See Figure 5.1

When a thread is executing, the `return_ip` field of the current frame is ignored; instead the GVMT handles the instruction pointer, `current_ip`. A thread is resumed by setting the `current_ip` to the `return_ip` of the current frame, then executing as normal. A thread is suspended by setting the `return_ip` of the current frame to the `current_ip`. Threads cannot be suspended in mid-bytecode.

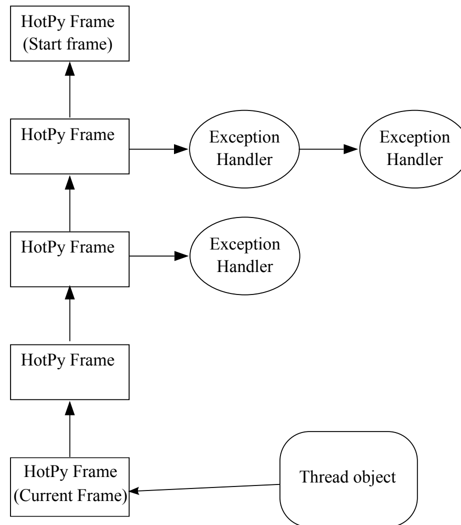


Figure 5.1: A HotPy Thread

Starting a Thread

Execution of thread is started by calling the `py_call` function. This sets up the current thread, pushing a new `frame` onto the frame stack, setting `current_ip` to the first bytecode in the called function and starting execution.

Bytecodes

Execution of a HotPy thread occurs by the successive execution of individual bytecodes. Each bytecode transforms the state of the VM. A complete description of all bytecodes in included in Appendix E.

Calling Functions

The `f_call` instruction expects three values to be on the data stack: the object to be called, a tuple of positional parameters and a dictionary of named parameters, with the dictionary on top of stack. The `f_call` instruction has varying semantics depending on the object being called.

When the callable object is a Python function, then execution proceeds as follows:

- The callable, tuple and dictionary are popped from the stack.
- A new `frame` is created and pushed to the frame stack.
- The frame is then initialised using the parameters stored in the tuple and dictionary.
- The `return_ip` field of the current frame is set to the address of the instruction following the `f_call` instruction.

- The `current_ip` is set to the first bytecode in the called function and execution proceeds.

Figure 5.2 shows the changes to the frame stack.

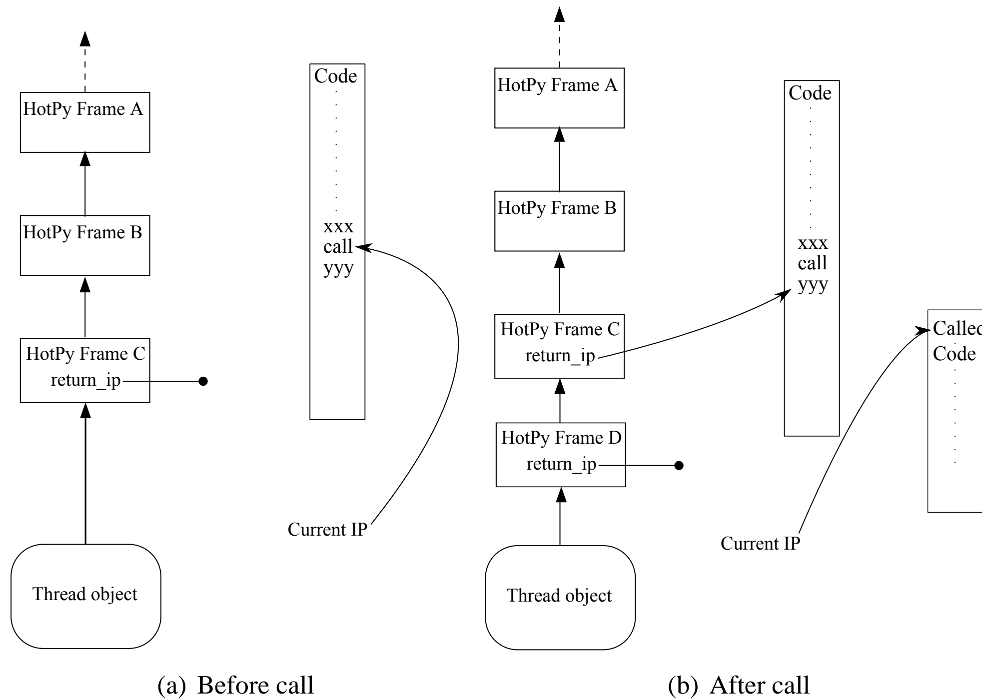


Figure 5.2: Call sequence

Calling native (C) code

In order to implement Python properly, particularly to support library code written in C, it must be possible to call C code from Python and vice versa. Calls to C code are effectively opaque to the VM. When calling a C function, the parameters are popped from the stack and passed to the function (this is handled by the GVMT).

C code may need to call back into Python code. For example, the `dict.__getitem__` method is implemented in C for speed, but may need to call the `__hash__` method of a class implemented in Python.

C code calls back into the VM, by calling a Python function using the `py_call` function. This creates a new HotPy frame, which is pushed to the frame stack, and calls back into the interpreter to resume execution.

5.3 Design of the HotPy VM

5.3.1 Overview

The HotPy VM is an advanced interpreter first and a compiler second. HotPy performs high-level optimisations as bytecode-to-bytecode transformations. These high-level optimisations, which are important for performance, are a key part of the VM design. Low-level optimisations, including compilation of bytecodes to machine codes, are handled by the GVMT-generated compiler.

The interpreter is in fact several interpreters in one. There is the main bytecode interpreter, a tracing variant of the main interpreter, a set of bytecode-to-bytecode translation stages (which are themselves bytecode interpreters) and a super-interpreter, which directs the execution of the various interpreters and of compiled code.

Execution of a program starts in the super-interpreter, which immediately calls the main bytecode interpreter to commence interpreting the bytecodes. When a back-edge or a call is encountered sufficient times, the super-interpreter checks to see if the code has already been optimised. If optimised code is found then the optimised code is executed, otherwise the tracing interpreter is started. Once the tracing-interpreter completes, the recorded trace is transformed to optimised bytecodes.

Tracing and optimisation may also be triggered when an exit from a trace is executed sufficient times. In this case the optimisation passes can use type information recorded at the exit point to generate better optimised bytecodes. HotPy uses trace stitching, as described in Section 2.4.3, to form traces over the working set of the program being executed.

5.3.2 Disconnecting the Two Machine States

The most straightforward implementation of HotPy would be to map the HotPy VM state directly onto the GVMT abstract machine state. In other words, function calls in HotPy would be implemented with calls at the abstract machine level, and exceptions would be implemented directly using the GVMT exception mechanism. The GVMT Scheme implementation in Section 4.10 shows that this works well and gives reasonably good performance. However, for an advanced optimising VM like HotPy, it is rather limiting.

Since HotPy is a *tracing* interpeter it must be able to execute traces, that is, it must be able to execute sequences of code whose structure is only weakly related to that of the original program. Traces may start or end in the middle of a function, cross function boundaries, or even end in the middle of a loop. The GVMT abstract

machine does not directly support this behaviour, so it is necessary to separate the HotPy VM state from the GVMT abstract machine state. It also happens that this separation of states makes the implementation of features such as generators and closures much simpler. It is assumed that any inefficiencies resulting from the separation of states causes can be removed by later optimisations. This appears to be the case in practice.

In order to separate the GVMT abstract machine state from the HotPy VM state, it must be possible to make arbitrary calls in one state without affecting the other. Similarly, the exception stack (try-except blocks) in Python should be unrelated to the GVMT state stack. This is achieved by implementing the HotPy call stack on the heap and by implementing exception handlers as objects attached to the current frame. Implementing the HotPy stack frames as heap objects makes it much easier to support generators, closures, debugging, and exception handling. Once HotPy frames are implemented as heap objects, it is straightforward to implement exception handlers for a try-except block as a linked list of handlers attached to the current frame.

Both trace exits and raising of exceptions are handled using the `gvmt_transfer()` function[1] which allows values on the data stack to be preserved. Finally, it is necessary to ensure that the depths of all the GVMT stacks remain bounded (and ideally, small) whatever the execution path. The super-interpreter ensures this while managing trace selection and execution.

Necessary Invariants

Whilst it is desirable to completely decouple the state of the HotPy VM and the GVMT abstract machine, it is not entirely possible. The problem is that if stack depths are totally unrelated it would be possible for a loop containing calls at the VM level to create deeper and deeper stack depth at the abstract machine level. To prevent this, a single invariant is required; at no point during execution may the abstract machine stack be deeper than when the currently executing VM frame was first executed. In other words, a return at the VM level *must* cause a return at the abstract machine level, if the corresponding call at the VM level caused a call at the abstract machine level. Also, no loop may be transformed into recursion.

5.3.3 The Super-Interpreter

The super-interpreter is a high-level interpreter which is concerned with executing sequences of bytecodes, rather than individual bytecodes. Its role is to dispatch the execution of code rather than perform that execution. Execution of bytecodes is the role of the interpreter or compiler-generated machine code.

[1]The gvmt_transfer() function is essentially an exception raising mechanism that, unlike gvmt_raise(), does not restore the data stack to its original value.

The super-interpreter performs the dispatch by looking up the current VM instruction pointer in a cache of traces, implemented as a hashtable. Each thread of execution has its own trace cache. When a trace is found in the hashtable, that trace is executed. If the trace is not found, then the unoptimised code is executed by the standard interpreter.

The super-interpreter is also responsible for handling exceptions. When an exception is raised it is caught in the super-interpreter. The super-interpreter then dispatches to the appropriate bytecodes for the exception handler.

The flow of control between the super-interpreter and the other interpreters can happen in both directions. Control can re-enter the super-interpreter when a trace finishes or when an exception is raised. The GVMT provides two mechanisms for a callee to pass control back to caller: a conventional return, and the `gvmt_transfer()` function. In HotPy, the former is used when execution moves from optimised trace to another, the latter for exception handling and other circumstances.

As execution progresses traces are recorded and optimised. Eventually a steady state should be reached where the vast majority of code executed exists in the cache of traces, as optimised traces. In this steady state the job of the super-interpreter is simply to execute one optimised trace after another.

Reenterant Super-Interpreter

The super-interpreter is the entry point to Python code from C code. Ideally, the only entry point would be at the start of the program, but a number of functions that must be implemented in C can call into Python code. Therefore the super-interpreter must be re-entrant. Since the super-interpreter is implemented on the GVMT abstract machine, its call depth does not correspond to the HotPy VM call depth, especially when handling exceptions, which may need to pass through an arbitrary number of super-interpreter invocations.

All activation frames must have a known call depth, both to conform to Python semantics and to prevent stack overflow. When the super-interpreter is invoked, it records the current call depth. Whenever the super-interpreter captures an exception, it checks to see if the call depth of the frame to be resumed has a depth that is less than the recorded call depth. If it does then the super-interpreter raises an exception to pass responsibility to the next outer invocation of the super-interpreter.

Figure 5.3 shows an example call stack for the HotPy VM. Each call to the super-interpreter corresponds to one or more calls at the VM level. The VM stack is maintained on the heap. Handling of calls at the VM level is simple enough; when a call is made in frame ‘B’, a new frame ‘A’ is created. Likewise returns are fairly simple: the top VM frame is discarded, and if the current frame was the entry frame for the super-interpreter, it returns as well. Exception handling is

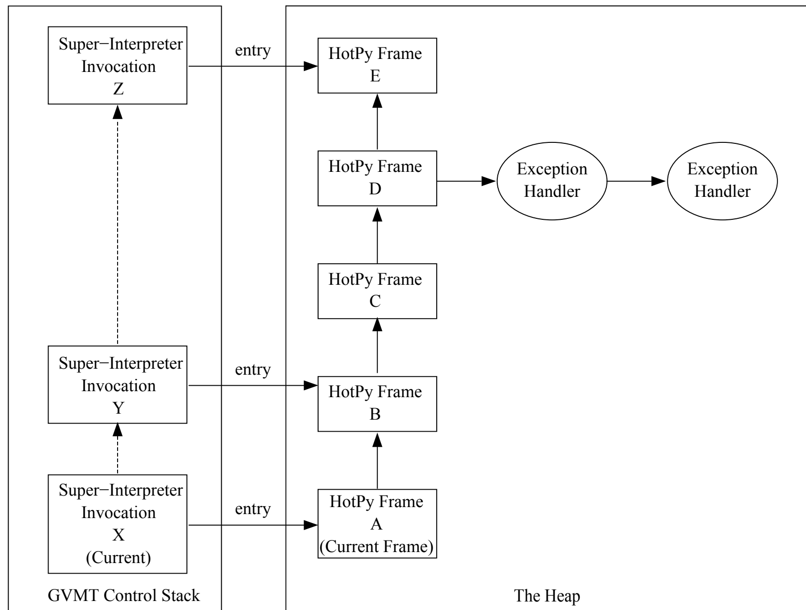


Figure 5.3: The HotPy Stack

more complex. Exception handler objects, which record enough state information to restore the VM state, are attached to VM frames. When an exception is raised, it is caught in the super-interpreter, which then checks the current frame for exception handlers, popping frames until it finds one. If it reaches its entry frame, the exception is re-raised, to be caught by the next outer super-interpreter. In Figure 5.3, if an exception were raised, it would be caught in the super-interpreter (`X`), which would re-raise the exception. This would then be caught by `Y`, unwinding the HotPy stack to reach frame `D` which contains exception handlers.

5.3.4 Active Links

Active links serve as the glue between traces (Figure 5.14 shows links joining traces). They are called *active* links as they can change their behaviour, but maintain the same interface to the rest of the VM. Active links allow the optimisation of traces, without changing the shape of the trace graph.

An active link consists of a pointer to a function, `call`, a pointer to some bytecodes, `ip`, a pointer to a trace (which is initially null), and some type information. The `ip` points to the unoptimised bytecodes. The function type of `call` takes two parameters; the first is the active link itself, which allows it to be self modifying; the second parameter describes the VM state. Active links embody a position in the bytecode, with type information to allow better specialisation.

The behaviour of an active link varies depending on how many times it has executed; how ‘hot’ it is. Cold code is executed, unoptimised, by the interpreter, in which case the `call` passes the original unoptimised bytecodes to the interpreter.

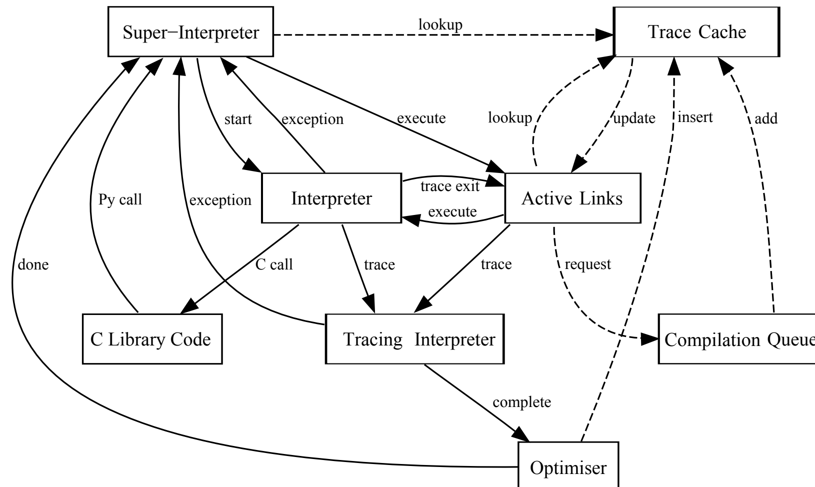


Figure 5.4: Control of Execution in HotPy

Warm code is still interpreted, but in an optimised form; `call` executes the interpreter with the optimised bytecodes in the trace. Hot code is compiled to machine code, which is executed directly; `call` executes the compiled code directly. All exits from traces point to active links, which are initially cold.

The self-modifying behaviour allows other code to treat active links as black boxes. The value returned by the `call` is the *next* active link to be run. This allows the steady state dispatching in the super-interpreter to be implemented as a simple call-threaded interpreter (Section 2.2.1):

```
do {
    link = link->call(thread_state, frame, link);
} while (1);
```

An active link can be in one of six states. Four of these states are starting states and depend on the instruction that caused the trace to exit, whether it was a boolean test failure, a back-edge, a return (or yield) or a many-valued test failure. These different states determine how aggressively the code is optimised and whether or not the starting context is used in specialising the trace. The two remaining states are interpreted traces and compiled code.

5.3.5 Control within the HotPy VM

Figure 5.4 shows the relationship between the components of the HotPy VM. Solid arrows represent control flow; dashed arrows represent data flow. As can be seen from the figure, active links perform a central role in the HotPy VM. Rather than explain each arrow, it is illustrative to consider the lifetime of an active link. In the following elaboration, words in *italics* correspond directly to labels in the figure.

An active link is created for each exit in a trace. Initially it is cold. Once it becomes warm it performs a *lookup* in the trace cache for a matching trace. It is probable than none will be found, so the active link starts a *trace*, which runs until it is *complete*, at which point the trace is optimised. The optimised code is *insert*ed into the trace cache. Control then returns to the super-interpreter once optimisation is *done*.

The active link will probably be executed again and will *lookup* the trace which, having been previously created, it will find. The trace is then *execute*d in the interpreter. If a trace raises an *exception* then control returns to the super-interpreter, otherwise a *trace exit* must occur and another active link gains control.

Once an active link becomes hot it sends a *request* to the compilation queue and continues to *execute* in the interpreter. Once compilation is completed, the compiled code is *add*ed to the trace in the cache, which *update*s the active link.

5.4 Tracing and Traces

The HotPy tracing interpreter is generated by the secondary interpreter generator, GVMTXC, from the same source as the main interpreter[2]. When tracing, HotPy records the values of the inputs and outputs of all instructions, as well as the intructions themselves, in order to generate a trace that is specific not only to the instructions executed, but also to the types of the variables used.

Tracing is initiated either by the interpreter or by an active link. The interpreter starts tracing when it detects a warm back edge or function call. An active link attached to an exit from the trace will start another trace when that exit becomes warm.

Traces started by the interpreter have no contextual type information and the resulting trace can always be executed in place of the original bytecodes. When tracing is started by an active link, the trace is specialised according to the type information recorded in the active link. This means that the trace can be better optimised, although the trace can be used in fewer contexts. This may produce more traces than just using non-specialised traces, as traces can overlap. Since traces tend to be very small relative to overall memory use, some duplication is not a problem. HotPy is designed so that the output from the tracing interpreter is executable bytecode; however, this is important only for testing and debugging, as traces are usually specialised and optimised immediately upon completion.

Finally, it should be noted that there is a one-to-many relation between points in the original bytecode and traces; a single bytecode may correspond to the start of several traces, all specialised for different contexts. This is illustrated by the example in Section 5.10.

[2]Making heavy use of #ifdef statements in the source code.

5.4.1 Recording a Trace

When recording a trace the interpreter must also perform the normal actions for that bytecode. Bytecodes can be classified as atomic, branching or non-atomic.

When the tracing interpreter encounters an atomic bytecode, it will record that bytecode. When a branching bytecode is encountered, an exiting equivalent is recorded. For example the branching bytecode `on_true` which jumps if the top of stack evaluates to `True` would be recorded as `exit_on_false` should the branch be taken. The exits from traces are discussed in Section 5.4.3. When a non-atomic bytecode is encountered, it may be recorded directly or a call to a surrogate function may be traced instead, Section 5.4.2 describes this in detail.

Halting

Tracing must halt, or the program could not terminate. HotPy halts tracing if any of the following conditions are reached:

- A loop is detected.
- A back edge is reached.
- Recursion is detected.
- More `return` or `yield` instructions are encountered than calls.
- An exception is raised in the trace or C code called from the trace.

In the first four cases the trace is kept for further optimisation. If a loop was detected, then the trace is closed; it can be executed immediately. If a back edge or recursion is detected, then the trace is saved and tracing continues with a new trace; it is assumed that a loop will be found in a subsequent trace. If too many `return` or `yields` are encountered then an unconditional exit is added to the trace after which the trace is saved and normal execution resumes. If an exception is raised then the trace is discarded and normal execution resumes.

5.4.2 Non-Atomic Bytecodes

In Python, and other high-level languages, the bytecodes have a high semantic content and are often non-atomic. A non-atomic bytecode is one whose execution state can be observed from the VM state. For example the bytecode `binary` may call an `__add__` function written in Python. The VM state can be interrogated during that function, even though `binary` is part way through its execution. Most bytecodes are atomic. For example, a `native_call` bytecode is atomic as the transfer of control occurs at the end of the bytecode.

The existence of non-atomic bytecodes complicates tracing since it may be possible to observe the VM in a state that corresponds to the middle of some bytecode.

In the case of the `binary` instruction, it is desirable to trace through any called function. However, that cannot be done if the `binary` bytecode is recorded, as the function that would be traced occurs in the middle of the `binary` instruction. One approach would be to code `binary` as a series of lower level bytecodes, each of which is atomic. However, for non-traced code and cases where the addition is performed by C code, this is grossly inefficient. HotPy gets around this by substituting, when tracing and where necessary, a Python function for the non-atomic bytecode.

Some bytecodes are non-atomic, but extremely unlikely to occur in a trace, such as `make_func`. These are recorded as normal; tracing is suspended during their execution to prevent incorrect duplication. The following bytecodes are non-atomic *and* likely to occur in a trace: the operators `binary`, `unary` and `inplace`, and `f_call`. The `f_call` can be atomic if it is calling a function, but non-atomic if calling a class.

In the case of the operators, normal lookup is performed. If a C function is found, the original bytecode is recorded. If no C function is found, the bytecode is not recorded and a Python function that performs the look up is traced instead. This is done in the expectation that, when optimised, the advantages of inlining the Python implementation of the operator will outweigh the penalty of the extra bytecodes.

In the case of calls to a function or bound method a special bytecode to mark the call site is recorded. Calls to classes are handled by looking for a surrogate Python function for the class, which is then traced. If no surrogate function is available, a more general Python equivalent of the `type.__call__` method is traced.

To see more clearly the problems here, consider the expression `d + e` where `d` and `e` are both `Decimals`, a standard library class written in Python. If the tracing interpreter were to record the `binary` bytecode and continue tracing, it would then record the body of the `Decimal.__add__` function. When this trace was executed it would execute the addition twice, once for the `binary` bytecode and once for the recorded call. So recording the `binary` bytecode and continued tracing are mutually incompatible. Since it is desirable to continue tracing, the `binary` bytecode cannot be recorded, but some semantically equivalent code must be traced instead.

The Python equivalents for the `binary` bytecode and `type.__call__`, along with the surrogate function for `tuple()`, are shown in Appendix D.

5.4.3 Trace Ends And Exits

When tracing reaches a conditional bytecode, the taken branch is recorded. However, when the trace is executed again a different branch could be taken. To handle this possibility conditional side exits are added to the trace. An unconditional exit

may be added to the end of the trace when tracing halts. These exits from traces are classified as follows in HotPy:

Back Edges

Tracing stops at back edges, in order to prevent non-termination of traces and to attempt to find loops. Consequently, on reaching a back edge during tracing, the trace is optimised, and a new trace is started immediately using the current context.

Return and Yields

Tracing normally continues through returns and yields, unless the trace is unbalanced, in which case tracing is halted. A trace is unbalanced when there would be more returns or yields than calls, were the trace to be continued.

Boolean Exits

When a conditional branch, such as an `if` statement, is encountered the tracing interpreter will record the taken branch only. An exit point must be inserted for the branch that is not taken. These side exits start tracing when they become warm.

Multi-Choice Exits

When a function or method call is encountered the tracing interpreter will record the function called and trace the execution of that function. An exit point must be inserted to handle cases where a different function is encountered during subsequent execution.

5.4.4 Avoiding Code Explosion

In Python, function or method calls are resolved dynamically. This means that a call site could potentially call a different function or method every time it was reached. Therefore exit points for function or method entry could potentially start a new trace every few iterations. This problem of code explosion is common for any form of specialisation; the number of specialised forms may grow almost without limit.

Informal measurements of the Self system showed that call sites call the same function about 93% of the time, two different functions about 5% of the time, and

more than two functions less than 2% of the time [37].

Assuming similar behaviour for Python, it would seem that the best approach for call sites that call more than two different functions is to simply resume normal interpretation.

HotPy avoids code explosion as a result of tracing by resuming standard interpretation if the trace exits at a call site. A more advanced approach would be to allow one, but only one, new trace at an exit. This would cover the cases where a call site calls two different functions. The current, simple approach seems to work well enough in practice.

5.5 Optimisation of Traces

Once a trace has been recorded it can be further optimised. All the HotPy optimisations, except the JIT compiler, are implemented as bytecode-to-bytecode translations.

Python is a highly dynamic language, which means that there are many events that could potentially occur during the execution of a Python program. Most of these events do not occur in most programs; they could, but in practice they do not. For example, function definitions bind a function object to a global variable; potentially a new value could be assigned to this variable, but this is unusual.

All optimisations in HotPy are focused on making the program fast in the case where these events do not occur, with little or no regard to program performance in the rare case that they do occur. However, program behaviour, ignoring timing and memory usage issues, must remain the same whatever optimisations are applied.

As an example of an optimisation, the name `list` in the global namespace usually refers to the list class; it could be redefined in a program, but this is regarded as bad practice. HotPy attempts to make the cost of reading of a value like `list` as close to zero as possible, even if this makes the cost of writing such a value very expensive.

5.5.1 Optimisation Chain

The optimisers in HotPy form a chain; once the tracing interpreter has completed recording a trace, it is optimised. The optimisers in HotPy are designed to work in a strict order, although individual passes can be omitted for experimental and debugging purposes. The order is: specialisation, deferred object creation, peephole (clean up), and finally compilation. Figure 5.5 shows the optimisation chain.

Specialisation is performed first as it requires the type information that is recorded

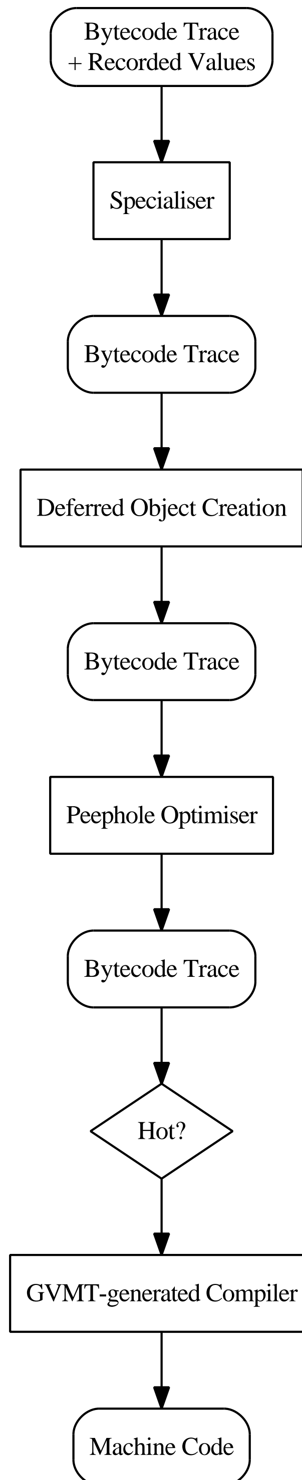


Figure 5.5: The HotPy Optimiser Chain

during tracing, and it makes the subsequent passes more effective. Specialisation replaces general bytecodes with type-specific versions. The Deferred Object Creation (D.O.C.) pass is next and removes redundant code that would otherwise create unnecessary objects. The peephole optimiser replaces short simple sequences of bytecodes with faster equivalents. Should the trace become sufficiently hot it is compiled to machine code.

Specialisation, along with tracing, is a speculative optimisation. In other words, it makes assumptions about the running program, so that it can better optimise the bytecodes. The other passes are conservative; they make no assumptions.

5.5.2 Guards

For HotPy to make assumptions about the running program, there must be some way to ensure that the program executes correctly if these assumptions are wrong. To do this HotPy must add extra code, known as guards, to ensure that any assumptions are either correct or, if they are incorrect, that a different code path is executed. HotPy uses two types of guard, inline guards and 'out-of-line' guards.

Inline Guards

The optimised code, produced by specialisation, will only work correctly for values of a particular class or even for a particular value. A guard is thus inserted immediately before the specialised operation; this is an inline guard. The instructions `ensure_tagged` and `ensure_type` used in the example in Section 5.10 are inline guards.

Out-of-Line Guards

Some operations, such as reading a global variable that is really a constant, are very common. Most of the work done in hashtable searches is unnecessary, endlessly rechecking the same values. In HotPy, and in CPython, changing a global variable or a class attribute involves a procedure call (in the VM, not in the Python program). Since these values are not expected to change, these procedures can be modified to include guards. The amortised cost of the guards is near to zero as the procedures are never called in practice, yet they allow the removal of the repeated checks from the frequently executed code. These guards are called out-of-line guards to differentiate them from inline guards, which must be executed whenever the guarded code is executed. Out-of-line guards are used in the example in Section 5.10, but are not visible in the trace.

Although the term out-of-line guard is new, the concept is not. The original Self compiler included the ability to invalidate code if certain assumptions were vio-

lated. The HotSpot JVM treats non-final classes as final, invalidating compiled code if a new subclass is loaded. Both of these features can be regarded as out-of-line guards.

5.6 Specialisation

The tracing interpreter records both the instructions executed and the values encountered during execution. It is reasonable to assume that the next time a piece of code is executed, it is likely to see the same types of values as the previous time it executed. This 'type stability' can be exploited by specialising the code so that it runs faster for the expected types of values. For example, if tracing records that the operands of a `binary` instruction are both tagged integers, the `binary` instruction is converted to an `i_add` instruction and two guards are inserted to ensure that both operands are tagged integers.

Specialisation, as this is known, is conceptually straightforward; specialised bytecodes are substituted for general bytecodes. In practice, the specialisation pass has also to insert guards to ensure that the code acts correctly if the types of values change later. If unexpected types are encountered then the trace exits. Even though specialisation is a fairly simple process, it can yield significant performance benefits, as shown in Section 6.5.

In HotPy, the specialiser performs all the optimisations that require the type information gathered by the tracing phase. This includes not only obviously specialising transformations such as converting a (general) `unary` operation to a (specific) `native_call`, but all other optimisations that depend on the type of the values expected. For example, the `load_global` instruction is translated to a `fast_load_global` or to a `fast_constant` in this pass, as type information is required to decide whether to treat the global as a constant or as a variable.

The specialiser also performs optimisations on data access, both global variables and attributes of classes and objects. These optimisations depend on the HotPy dictionary structure and are discussed in Section 5.12.2.

The trace produced by the specialisation pass has type information embedded in it, in the form of `ensure` guard instructions and specialised operations such as `i_add`. This allows subsequent optimisation passes to act on the bytecodes in a trace without requiring additional type information. All speculative optimisations are performed by the tracing pass (customising for flow control) and the specialiser (customising for type). Later optimisations are not speculative, taking advantage of the opportunities exposed by the speculative passes.

5.6.1 Specialisation of Bytecodes

Specialisation is a linear pass operating on one bytecode at a time. The specialiser maintains type information about local variables and the stack. It uses this information, combined with the type information recorded during tracing to specialise individual bytecodes.

Specialisation of a bytecode is a five stage process:

- Look up the types of the operands of an instruction.
- Add guards to ensure the type or other property of any operands required to specialise the bytecode.
- Update the type information for the operands.
- Emit the specialised bytecode
- Store the type information for the result of the bytecode.

For example, consider a `binary` addition bytecode. Now assume that the top two values on the stack have been recorded as tagged integers by the tracing phase. First of all the types of the operands are looked up and found to be *probably* tagged integers. Guards must be added to ensure that the operands are indeed tagged integers; two bytecodes, `ensure_tagged` and `ensure_tagged2` are emitted. The type of these values is now known, so the type information is updated. The specialised bytecode `i_add` is now emitted. Since the result of `i_add` is always a tagged integer, this information is recorded. In this example the `binary` bytecode is replaced with the sequence `ensure_tagged ensure_tagged2 i_add`. Although the new sequence is longer, the individual bytecodes are much faster.

For instructions like `load_global` the process is the same. The difference is that the guard required is an out-of-line guard, so does not show up in the resulting trace.

5.6.2 Recording Type Information

The type information for a value is recorded as a triple; a class object, a set of three boolean flags, and a dictionary-keys objects (for optimising the `load_attr` bytecode, see Section 5.12.2). The three flags record whether a value is definite or probable; whether or not it is a tagged value; and whether it is positive (it *is* the class) or negative (it is *not* the class). Negative types are required for exits where a guard has failed; the guarded value will be *definitely not* an instance of the class.

During specialisation, type information is recorded for the local variables of the current frame, for the local variables of all frames recorded in the frame stack and for all values on the stack. When type information is lacking for an operand, the type of the value recorded during tracing is used as the *probable* type.

Type information is recorded in the active links for all exits, to ensure effective specialisation of 'hot' exits. In order to avoid excessive specialisation only a limited amount of type information is recorded; the local variables of up to two frames and the top two values on the stack. To avoid excessive memory usage this information is stored in the active links in a compressed form.

5.6.3 Loop Optimisation

Loop optimisation in HotPy, like loop optimisation in a conventional compiler, consists of moving code out of the loop. On completion of specialising a loop, two sets of type information will be available, one for entry to the loop and one for exit from the loop. For correctness it is sometimes necessary to insert extra checks immediately before the end of the loop if the type of a variable is wider at the end of loop than at the start. Conversely, it is beneficial to insert extra checks before the start of the loop to narrow the type of any variables that are wider at the start of the loop than at the end, eliminating the need for checks in the loop. It is also possible that the types of a variable at the start and end of the loop are contradictory, in which case a check must be inserted at the end of the loop, to enforce the type expected at the start. This check will fail, causing the trace to exit. Hopefully, a type-stable loop will be found after some small number of additional traces.

5.6.4 Avoiding Code Explosion due to Specialisation

Code explosion can be caused by specialisation, as well as by tracing. The specialiser avoids causing code explosion by not specialising the first bytecode in a trace. This has a small cost, as traces are not as well specialised as they could be. However, it does ensure that no more than two different traces can result from specialising a bytecode.

5.7 Deferred Object Creation

Tracing and specialisation may expose redundancy in the form of parameter handling, checks around calls and in the form of repeated checks. The deferred object creation (DOC) pass can remove many of these redundancies.

The DOC pass implements a form of escape analysis in order to avoid creating expensive temporary objects. To conform to Python semantics, HotPy must create a lot of small objects which have a limited lifetime. Although HotPy possesses a generational garbage collector which allows such objects to be created cheaply, there is still a significant cost to allocating and initialising these objects.

Many of the objects have a lifetime of only a few bytecodes and exist only as temporary containers for other values. Most of these short-lived objects are created in order to manage the passing of parameters to procedures. Parameters are passed in tuples and dicts[3] and then stored in a frame. Frames, tuples and dicts are all heap-allocated objects. By deferring the creation of these objects it is often possible to avoid creating them at all.

Deferred object creation currently defers the creation of the following objects: tuples, (empty) dicts, bound methods, frames and slices[4]. For small functions, such as property get methods, that tracing has inlined, the DOC pass can reduce the code executed to a minimum. The DOC pass, like all the HotPy optimiser passes, is a linear-time pass.

5.7.1 Shadow Stacks

The DOC pass defers creating objects for as long as possible. To do this, it maintains a shadow data stack and a shadow frame stack to record objects that it is currently deferring. The shadow data stack and a shadow frame stack record the difference between the original, non-deferred state and the actual, deferred state. When the DOC pass encounters an instruction that would create a new object which would be of a type that the DOC pass understands, such as tuple, then a deferred object is pushed to the shadow stack. The DOC pass also maintains a shadow line number.

There are a number of instructions that the DOC pass understands but cannot defer. To handle these, the DOC pass is able to mix objects that have already been created with deferred ones. For example, if the DOC pass encounters an `i_add` instruction it must ensure that the top two values on the stack actually exist, emitting the code to create any deferred objects. It then emits the `i_add` instruction and pushes a marker to the deferred stack, showing that the object on top of the shadow stack corresponds to the one on top of the real stack.

5.7.2 Thread-Local Cache

In order to handle a stack of mixed deferred and non-deferred objects, a thread-local cache is required to store non-deferred objects. These cached objects can then be treated as deferred objects; the deferred operation is the operation of loading them from the cache. For example, the `pack 2` instruction takes the top two values from the stack and creates a tuple. The DOC pass wants to defer creation of the tuple. This is easy if the top two values on the stack are deferred, but what

---

[3] In Python the built-in dictionary type is known as 'dict'.

[4] See `http://docs.python.org/library/stdtypes.html` for a description of these data structures.

if one is not? Suppose that the object on the top of stack is a deferred constant, but the second object on the stack is a real, non-deferred object. In this case the DOC pass emits a `store_cache` instruction to move the real value off the stack into the cache. The deferred tuple then consists of a pair of deferred objects: the deferred constant and a deferred load from the cache.

5.7.3 Reconstruction of the Original VM State on Exits

In order for deferred object creation to work effectively, it must defer the creation of objects quite aggressively. To successfully do this and to maintain correctness, sequences of code to restore the VM state must be added to all trace exits. Furthermore, in the case of native calls that do not modify global state, but may raise an exception, code to restore the state must be attached to exception handlers across such calls. In the example above, the `native_call` instruction is converted to a `native_call_protect` instruction. The `native_call_protect` instruction attaches corrective code to the thread exception handler for the duration of the call. In the event of an exception being raised, the corrective code is executed, which recreates the correct VM state.

Since the generation of these code sequences is done once during optimisations, while the savings due to not creating objects unnecessarily occur continuously, the potential speed up is significant.

5.7.4 An Example

The following example is taken from the gcbench benchmark used in the next chapter.

Figure 5.6 shows snippets of source code which are covered by a single trace during tracing. The first two snippets, line 87 and lines 52 - 56, are from the gcbench program; the third snippet, lines 77 - 80, is from the HotPy library.

Execution of the trace starts by calling the class `Node` to create a new instance (first snippet, line 87). This has been traced through the library code for object creation (third snippet, lines 77 -80), which calls the `__init__` method of the `Node` (second snippet, lines 54-56).

The program is run with the DOC pass turned off and the trace in Figure 5.7 is created. The numbers on the left are the offset from the start of the trace, in bytes. All hexadecimal values are the addresses of objects that have been pinned and inlined by the specialiser. All instructions of the form `line_xxx N ...` set the line number to `N` and perform operation `xxx`.

When the program is run with the DOC pass turned on the trace shown in Fig-

```
87     return Node()

52 class Node(object):
53
54     def __init__(self, l=None, r=None):
55         self.left = l
56         self.right = r

77 def alloc_and_init(cls, *args, **kws):
78     obj = object_allocate(cls)
79     cls.__init__(obj, *args, **kws)
80     return obj
```

Figure 5.6: Source Code for DOC Example

ure 5.8 is produced. The new trace is is a third of the length (12 rather than 35 instructions) of the previous version. The DOC pass is a linear pass, so its actions can be followed by scanning the trace in Figure 5.7 from top to bottom. The DOC pass was able to remove two thirds of the code as follows.

The net result of the code from offsets 0 to 38 is to a create a new frame and push a constant value to the stack. The DOC pass defers these operations as neither the frame nor the value is required yet. For each instruction in the original sequence, the operation required is performed by the DOC on its shadow stacks, no instructions are actually emitted. Figure 5.9 shows the state of the shadow data stack and shadow frame stacks for each instruction in the sequence. The states shown are those *after* the instruction has been evaluated; the data stack is to the left of the | divider.

The `native_call` instruction at offset 40 requires parameters on the stack, so the stack can be deferred no longer and the DOC pass emits the `fast_constant` instruction to recreate the single constant value on the stack. Since the `object_allocate` function is annotated as *not* accessing global state, there is no need to create the frame across the call. Since line numbers are stored in the frame, if the frame is deferred then the `line` instructions can be deferred as well.

The `store_frame` instruction at offset 46 stores the newly created `Node` into the current frame. However, the current frame does not exist, having been deferred, so the DOC pass stores the value into the thread-local cache, emitting the `store_to_cache 0` instruction.

So far the DOC pass has consumed 14 instructions, emitted three and has deferred the creation of a frame.

The instructions from 56 to 77 marshal the parameters for the `Node.__init__` function and then uses them to create a new frame. Once again the DOC pass defers creation of the new frame. There are now two frames on the shadow frame

```
0       :line_fast_constant 87 0xb7b0afa0
7       :empty_tuple
8       :dictionary
9       :new_enter 0x82aa080     /* Entry to alloc_and_init */
14      :make_frame 2 0xb7b0a4e4
20      :init_frame
22      :line_fast_constant 78 0x82aa800
29      :fast_constant 0xb7b0afa0
34      :pack_params 1
36      :drop
37      :drop_under
38      :unpack 1
40      :native_call 0x80d5630 /* Call to object_allocate */
46      :store_frame 3
48      :line_fast_constant 79 0xb7b0afa0
55      :drop
56      :fast_constant 0xb7b0aec8
61      :fast_load_frame 3
63      :pack 1                  /* Parameter marshalling */
65      :fast_load_frame 1       /* on line 79 */
67      :tuple_concat            /* ditto */
68      :fast_load_frame 2       /* ditto */
70      :copy_dict               /* ditto */
71      :make_frame 2 0x828f4cb /* Entry to Node.__init__ */
77      :init_frame
78      :line_fast_load_frame 55 1
82      :fast_load_frame 0
84      :fast_store_attr 4 4   /* self.left = l */
89      :line_fast_load_frame 56 2
93      :fast_load_frame 0
95      :fast_store_attr 4 1   /* self.right = r */
100     :func_return
101     :line_fast_load_frame 80 3
105     :func_return
106     :return_exit 0xb7b109c0
```

Figure 5.7: Trace Without DOC

```
0       :fast_constant 0xb7a8afa0
10      :native_call_protect 0x80d5630  0xb7b07908
16      :store_to_cache 0
18      :none
19      :load_from_cache 0
21      :fast_store_attr 4 4   /* self.left = l */
26      :none
27      :load_from_cache 0
29      :fast_store_attr 4 1   /* self.right = r */
34      :load_from_cache 0
36      :clear_cache 1
38      :return_exit 0xb7a90fbc
```

Figure 5.8: Trace With DOC

```
0  line_fast_constant:          Node | <empty>
7  empty_tuple:             Node, () | <empty>
8  dictionary:          Node, (), {} | <empty>
9  new_enter:  alloc_init, (Node,) {} | <empty>
14 make_frame: alloc_init, (Node,) {} | (locals = -, -, -)
20 init_frame:                        | (locals = Node, (), {})
22 line_fast_constant:     obj_alloc  | (locals = Node, (), {})
29 fast_constant:    obj_alloc, Node  | (locals = Node, (), {})
34 pack_params: obj_alloc, (Node,), {}| (locals = Node, (), {})
36 drop:           obj_alloc, (Node,) | (locals = Node, (), {})
37 drop_under:                (Node,) | (locals = Node, (), {})
38 unpack 1:                     Node | (locals = Node, (), {})
```

Figure 5.9: Shadow Stacks For Start of Trace in Figure 5.7

stack. The instructions at offsets 78 to 84 load two local variables and store one into a slot in the other. The DOC pass can defer the loads, but cannot defer the `load_slot` instruction so is forced to materialise the stack (but not the frames). The DOC pass materialises the stack, consisting of `None` (the default parameter for `l`) and the new `Node` object, by emitting `none` and `load_from_cache 0` before the `load_slot` instruction.

The `func_return` instruction at offset 100 pops the topmost deferred frame. The second `func_return` pops the remaining deferred frame. The `return_exit` instruction forces the DOC to recreate the entire VM state. As there are no deferred frames, only the stack needs to be updated with a `load_from_cache 0` instruction. Finally, a `clear_cache 1` instruction is emitted so that the cache does retain any objects.

5.7.5 Unboxing Numbers

As well as avoiding the creation of composite objects, it also beneficial to avoid creating boxed numbers. Since HotPy uses tagged integers, this is not as important as it might be for other VMs, but HotPy does use boxed floats. The DOC pass also handles floats, but is limited in its effectiveness, as it can only unbox intermediate values that spend their whole lifetime on the stack. As soon as a float is stored in a local variable then that float must be created (un-deferred). Deferred object creation could be extended in two ways, both of which would improve the performance of floating-point numbers. The first would be to defer object creation across loop boundaries; the second would be to defer stores into non-deferred frames.

5.8 Further Optimisations

The remaining optimisations performed by HotPy are ‘peephole’ optimisations, that is they are simple transformations which replace a short sequence of instructions with a more efficient form. For example, tracing may insert a conditional exit immediately after a boolean test, say `exit_on_false`. Specialisation may then be able to infer that a comparison is always true, and convert the expression to a single `true` instruction. The redundant instruction pair `true`, `exit_on_false` can then be eliminated.

Other examples include making the instruction sequence more efficient for the GVMT’s stack-based compiler, such as replacing a `store_frame`, `load_frame` pair with a `copy`, `store_frame` pair. A few more complex replacements are performed, aimed at cleaning up the output from the main optimisation passes.

5.8.1 Compiling Traces

Once a trace becomes sufficiently hot, it is added to a priority queue for compilation. In order to prevent undue pauses, the compilation time is limited to a certain fraction of the execution time. Potentially compilation could take place in a separate thread from the interpreter, but this has not been implemented yet. On a single processor it is limited to approximately one quarter of total execution time. On a multi-processor machine compilation is limited, arbitrarily, to approximately one half of the execution time of the interpreter thread.

Traces are compiled using the GVMT generated compiler. The code generated by the compiler matches the interface of the interpreter[5]. This almost matches the signature of the `call` function in the active link (see Section 5.3.4) so immediately before compilation an extra instruction is inserted in front of the first bytecode to discard the extra parameter of the `call` function. The `call` pointer can then be set to point directly to the compiled code.

5.9 De-Optimisation

All the optimisations in HotPy are either speculative or depend on other speculative optimisations. These need to be guarded, as described in Section 5.5.2. When an inline guard fails, execution continues correctly on a different path. However, when an out-of-line guard fails, it invalidates code, which must never execute again. In order to prevent the execution of invalidated code, all traces are checked for validity before execution. In addition to the check at the start of a trace, a `deoptimise_check` instruction is also inserted after any call to code which might invalidate the current trace. Invalidated traces are unlinked from any active links that may attempt to call them. When they are no longer referenced, they are garbage collected.

5.10 An Example

Figure 5.10 shows a simple Python program for finding a list of Fibonacci numbers which will be used to illustrate how HotPy creates and links traces. Although a very small program, it serves to illustrate some of the key points of HotPy's operation. The print statement on the final line is commented out to prevent the traces becoming too large to show on a single page.

Compiling the source code into bytecodes gives the flow graphs for the functions `fib` and `fib_list` in Figures 5.11 and 5.12 respectively. The flow graphs show a

[5] Except that it does not take a bytecode address as its first parameter; see the GVMT manual for more details.

```
1   import sys
2
3   def fib(count):
4       n0, n1 = 0, 1
5       for i in range(count):
6           yield n1
7           n0, n1 = n1, n0 + n1
8
9   def fib_list(count):
10      return [f for f in fib(count)]
11
12  fibs = fib_list(int(sys.argv[1]))
13  #print(fibs)
```

Figure 5.10: Fibonacci Program

direct correspondence to the source code.

Running the program with an input of 40 causes the loop in the `fib_list` function to become warm, and HotPy starts tracing. Tracing is triggered when the execution count of `end_loop` instruction exceeds the threshold value. Tracing then starts from the next instruction to be executed, which in this example is a `load_frame` instruction. Tracing continues until the `end_loop` instruction is reached again and a closed loop is recorded.

During tracing, the call to the `fib` generator function is inlined; the resulting trace is shown in Figure 5.13(a). Entry to the `fib` generator function is marked by a `gen_enter` instruction. The `gen_yield` instruction marks the point where execution returns to the `fib_list` function. The `<ENTRY>` at the top of the trace means that the trace can be entered directly from the interpreter and corresponds to the `end_loop` instruction in the `fib_list` function.

Since the cost of bytecode-to-bytecode translation is low, the trace is immediately specialised to produce the trace shown in Figure 5.13(b) and then further optimised (in this example further optimisation has no effect). The resulting trace is specialised not only for the flow-control, which happens during tracing, but for the types of values observed during tracing. In this example, the `binary` (addition) instruction has been replaced with the `i_add`, which is specialised for tagged integers. Guards have been added to ensure correctness. All the (guarded) side exits are cold, as the program executes mainly around the loop. The optimisations are explained more fully in Section 5.5.

If the program is run with a higher input value, say 60, then `n0` and `n1` will exceed the maximum size for tagged integers and must be stored as boxed integers on the heap. This will cause one of the `ensure_tagged` guards in the loop to fail. Once it has failed a sufficient number of times, the side exit is then hot and HotPy

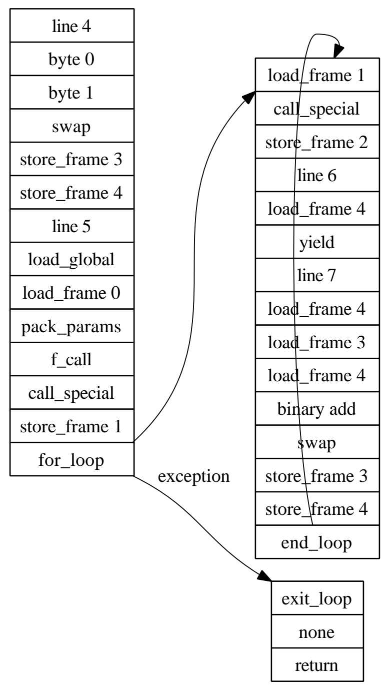


Figure 5.11: Flowgraph for fib function

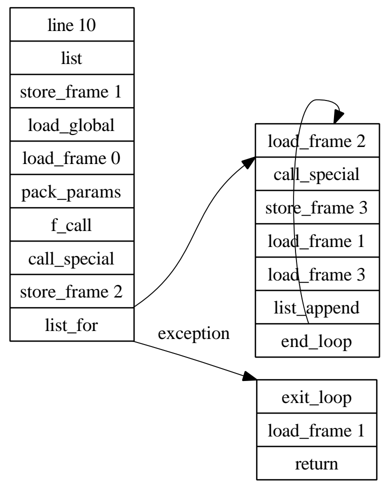
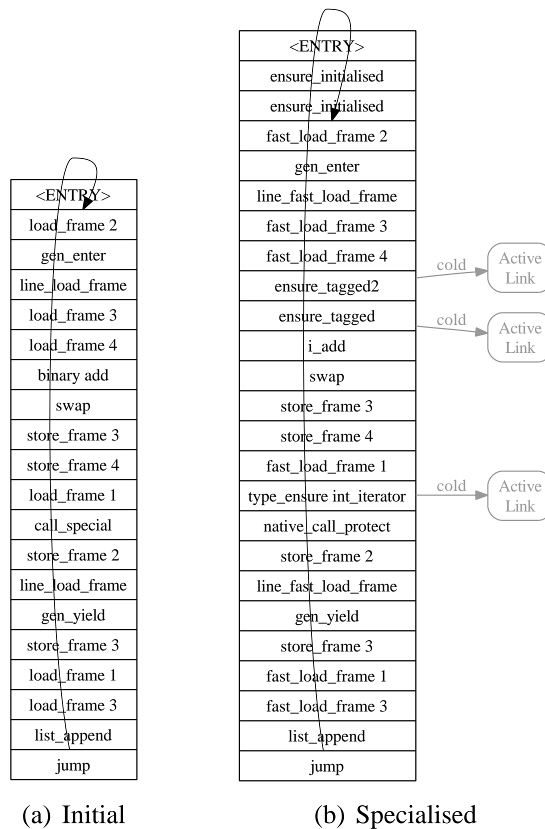


Figure 5.12: Flowgraph for fib_list function



(a) Initial (b) Specialised

Figure 5.13: Traces of the Fibonacci Program With an Input of 40

starts tracing from that point. HotPy maintains type information for each exit (in a compact form), meaning that when a new trace is recorded it can be more effectively specialised.

The resulting, optimised, trace graph is shown in Figure 5.14. As execution proceeds from the first hot exit, a new trace is created until a back edge is reached. In order to discover loops, tracing must restart on reaching a back edge. This causes the intermediate traces in the middle of Figure 5.14 to be created before a new loop is found, which is shown on the right of Figure 5.14. The new loop is almost identical to the original loop, but is specialised for boxed, rather than tagged, integers and does not require an `ensure_initialised` instruction on entry. The seventh to ninth instructions show the different specialisation.

The additional short, unconnected, trace in Figure 5.14 is caused by parts of lines five and six of the program becoming hot while the program transitions from the loop on the left to the loop on the right.

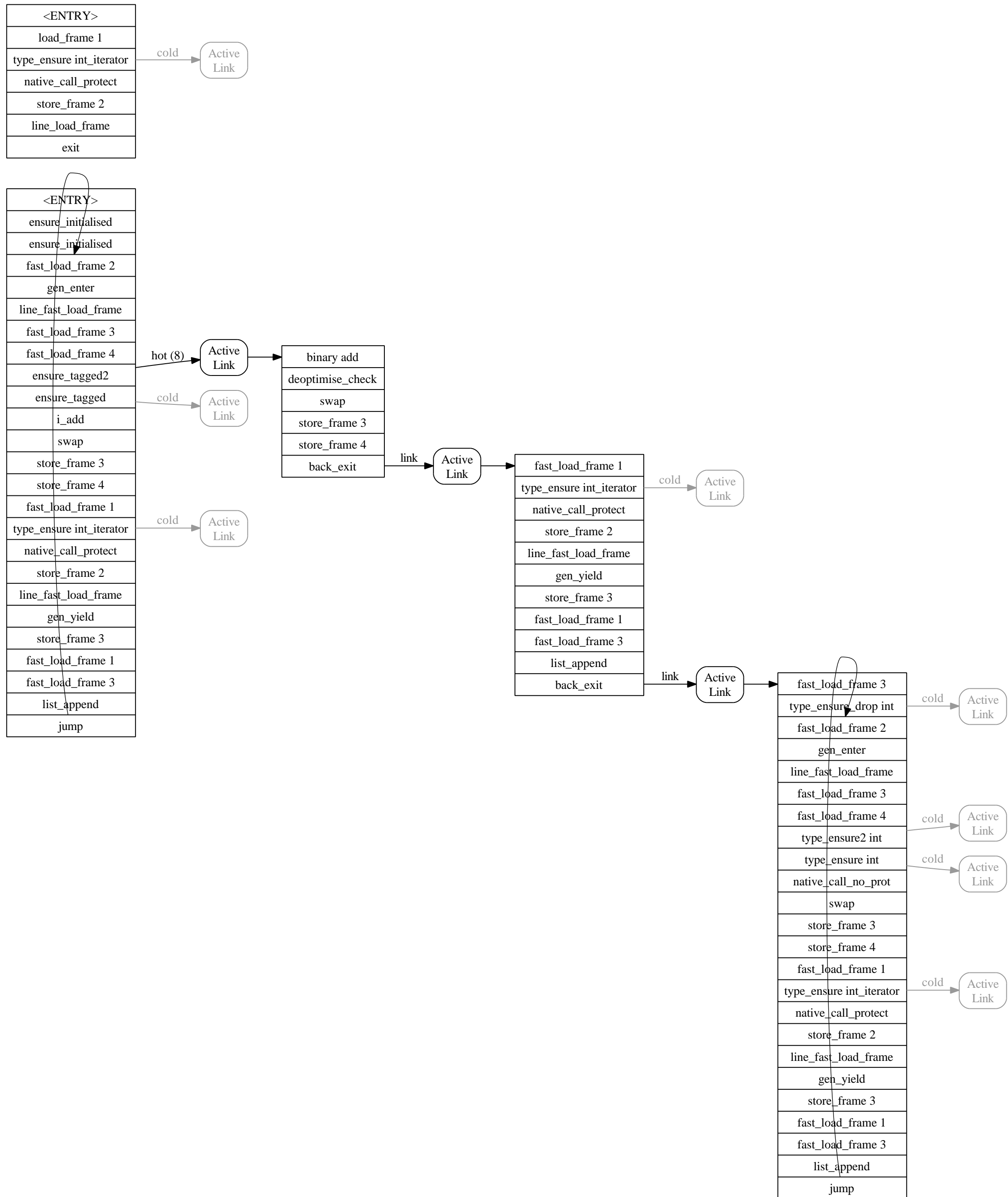


Figure 5.14: Extended Trace for Overflow

5.11 Deviations from the Design of CPython

Where possible, HotPy follows the overall design of CPython. However HotPy does differ in some notable ways. Apart from the obvious differences in optimisation of bytecode and JIT compilation, there are differences in the way classes are laid out, in the way that operators are handled, and in the implementation of the dictionary type.

5.11.1 Class Layout

A class (`type`) object in CPython contains over 70 pointers in order to support, reasonably efficiently, the large number of 'special' methods required by Python. In Python, a special method is one where the existence of an attribute with the same name in the object does not alter the behaviour, as it would for a non-special method. For example, any type that supports addition must have `__add__` and `__radd__` methods. A slot in the `type` object is required for each of the many special methods. Since the special methods are also visible to the Python programmer, a slot wrapper[6] object must be also be installed in the type's attribute dictionary for each special method.

HotPy dispenses with all but six of these pointers, storing the other 60+ special attributes directly in the type's dictionary. Five special-method pointers, `__getattribute__`, `__setattr__`, `__get__`, `__set__` and `__delete__` are necessary for correctness. One additional pointer `__hash__` is retained for efficiency. Although this simplification would be expected to reduce performance, in practice it has little effect, due mainly to the way that HotPy handles operators.

5.11.2 Operators as Partial Multi-Methods

In Python, the semantics of binary operators, such as addition, are defined in terms of dispatch on the operand types, firstly on the left operand, and then on the right operand. The semantics are complicated by subtyping, but in general work as follows: Consider the expression `x + y`. To determine the value of this expression, Python first evaluates `x.__add__(y)`, and should that fail, it then evaluates `y.__radd__(x)`. Both `__add__` and `__radd__` are special methods.

In CPython, addition is implemented by trying `x.__add__(y)` before `y.__radd__(x)`, with failure indicated by returning the `NotImplemented` value. For example, the expression `i + f`, where `i` is an `int` and `f` is a `float`, is evaluated by CPython as follows: CPython calls the function `__add__` belonging to `int`: `int.__add__(i, f)`. Since `int.__add__` can only handle

[6]A slot wrapper is an object installed in a class dictionary which makes the slots (pointers) in the C implementation visible to the Python programmer.

addition of ints this fails, returning NotImplemented. CPython then calls float.__radd__(f, i) which returns the correct result.

In HotPy, operators are implemented as partial multi-methods. Each operator contains a hashtable which maps all valid pairs of the built-in types to the relevant function. For example the add operator contains a mapping from the (int, float) pair to the add_int_float function. The operation x + y is computed by looking up the pair (type(x), type(y)) in the hastable and applying the resulting function to (x,y). If no function is found, CPython-style double dispatching is then used. Although slower for user-defined types, it is significantly faster for built-in types,. It also makes the tracing and optimisation of binary operations faster and simpler as only user-defined types need to be handled.

5.12 Dictionaries

In Python, dictionaries are used both as mappings in user code and to implement namespaces in the virtual machine. Python has three kinds of namespaces; type attributes, object attributes and global (module-level) variables. Type attributes are stored in a special type, dict_proxy, which is implemented as a standard open-addressed hashtable. However, object attributes and global variables are held in standard Python dictionaries, of type dict. This means that the dict class has to perform three similar but different roles; object namespace, module namespace and explicit mapping. Although the dict has only one interface, each of the three roles has distinct usage characteristics.

In CPython, the dict is implemented as an open-addressed hashtable that has been refined over several years. It is tuned for a combination of the usage characteristics of module variables and object attributes. This works well for CPython, but HotPy has different requirements and performance characteristics. For example, in CPython memory allocation is slow and this constrains the design of the dict. Memory allocation is considerably faster in HotPy, so allocation is not a bottleneck.

5.12.1 Python Dictionary Usage

Analysis of the usage of dicts in Python (the language, not any particular implementation) suggests a different design for the dict from that of CPython. The most common use of dictionaries in Python is not explicit, but implicit, as containers for global variables and object attributes.

Global Variables

Global variables in Python are often effectively constants such as functions or classes; not only do these (almost) never change, they can be read very frequently. Even if some variables do vary, the set of variable names, which is the set of keys in the `dict`, changes very rarely. In order to optimise reading of global variables, it is useful to be able to keep them in a known location in memory, so access can be fast. In order to treat the effective constants as actual constants it is necessary to guard against their values changing.

Object Attributes

Most objects of a given class, once initialised, will share identical attribute names. In other words, for any given class it is highly likely that `dicts` of all the objects of that class will have equivalent keys. Thus memory use can be cut in half by ensuring that, for those classes that allow it, all objects of a class share the same keys. This also means that the offset of any attribute in the objects of a given class are computable from the class alone. Unlike the values in a module's `dict`, the values in a `dict` used to hold an object's attributes *are* likely to change; it is just the keys that are unlikely to change.

Program-Level Mappings

Although this usage is less frequent than for global variables and object attributes, it is nonetheless important. Any optimisations designed to improve the performance of the above cases should not impact the performance of the explicit use of dictionaries too much.

5.12.2 Design of the HotPy Dict

Noting that allocation is not a bottleneck, and that object attribute dictionaries stand to gain from sharing keys, the main idea behind the design is to split the keys and values of a `dict` into two different objects, rather than pairing them. So instead of one table consisting of [key, value, key, value...] there are two tables: [key, key, ...] and [value, value, ...], the nth key corresponding to the nth value. In order to allow safe[7], concurrent resizing of `dicts`, the reference to the keys object must be stored in the values object, not in the `dict` directly. Additionally, shared keys must be immutable, or race conditions might occur. These constraints have a negative effect on access times to keys, but it is small compared to the benefits of the optimisations that become possible.

[7] Not race free, but ensuring the dict remains in a valid state.

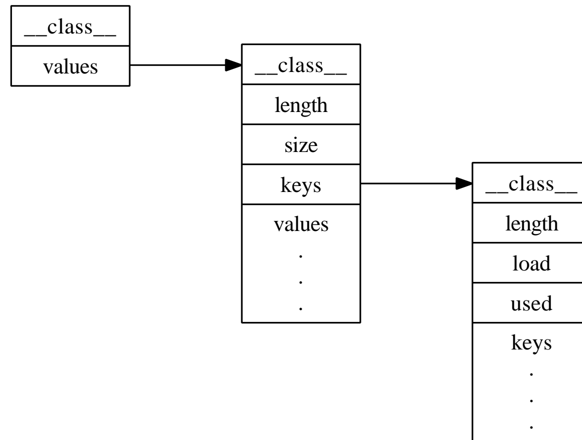


Figure 5.15: The HotPy dict

Figure 5.15 shows the HotPy `dict` implementation. In the values object, `length` is the length of the values array, `size` is the number of values, and `keys` refers to the keys object. In the keys object, `length` is the length of the keys array, `load` is the maximum number of keys before resizing, and `used` is the number of keys (some of which may have a corresponding nil in the values object, if the value has been deleted). Note the invariant $values.size \leqslant keys.used \leqslant keys.load$. By adding a further invariant that a key is never removed from a keys object[8] some useful optimisations are possible. To allow sharing of keys objects, shared keys objects are initialised with $keys.used = keys.load$. Combined with the constraint $keys.used \leqslant keys.load$ and the prohibition on removing keys from a keys object, this makes these keys objects immutable.

Finally, given that the values object is separate from the `dict` object, it is possible to give the values object of a module dictionary a different class, and thus different behaviour from that of a non-module `dict`.

Attribute Optimisations

In CPython and unoptimised code in HotPy, accessing an attribute of an object involves complicated, and thus slow, look up. The full semantics of Python attribute look up is described in Appendix C. The attribute being read may be an overriding descriptor, such as a property, a non-overriding descriptor, like a method, or an ordinary attribute, stored in the object's dictionary.

If an attribute is an overriding descriptor, it can be optimised by inserting a guard into the class's dictionary to ensure that the attribute does not change. This allows the descriptor's `__get__` method to be called directly, or possibly inlined.

[8] Keys can be removed from a dict by resizing it, possibly to the same size; unused keys are not copied during resizing.

If an attribute is stored in the object's dictionary, a more complex optimisation is required. It is worth recalling that non-descriptor attributes in Python are independent of the object's class. This makes object dictionaries in Python similar to objects in a prototype-based language, such as Self or Javascript. In Self, artificial class-like objects are constructed to group objects into something like classes. HotPy does something similar. Each class caches a keys object, which is used to initialise the `dict` of every object of that class. This ensures that for most classes, all objects with the same class will share the same keys object. As well as saving memory, this can be used for performance optimisation. During optimisation the offset of the key in the keys table is found, and this, as well as the offset of the dictionary in the object, can be used to perform fast attribute fetches and stores. An out-of-line guard must be inserted into the class (and into its super classes) to ensure that it does not acquire a descriptor of the same name as the attribute. An inline guard must be inserted to ensure that the keys object in the `dict` matches the keys object in the class. The address of the attribute in question is then `o->__dict__->values[key_offset]`. No dictionary lookups or class searches are involved.

If an attribute is a non-overriding descriptor, such as a method, similar guards to those above must be inserted to ensure that the object has *no* attribute in its dictionary which could mask the descriptor. The descriptor can then be called directly. In the case of a Python function, tracing may have already inlined the call.

Global Variables and Constants

In Python, classes and functions are bound to names in the same way as any other value. This makes it impossible to tell for certain whether a global variable is in fact a constant. The distinction between variable and constant is important. Treating a variable as a constant would result in wasted effort as code is optimised only to be discarded, but treating a constant as variable would result in considerably less efficient code. HotPy uses the very simple heuristic that global variables holding classes or functions are constants and all others are variables.

Since all global variables are kept in `dicts` belonging to modules, when these `dicts` are created they are given a values object with a different class from non-module `dicts`. This values object can hold additional guards to protect attributes against deletion and, in the case of values treated as constants, against modification. By pinning[9] the values object, the address of the global variable can be computed during optimisation and global variables can be accessed by a single read, as fast as a statically typed language. Constant values can be inlined into the bytecode.

[9]Preventing the garbage collector from moving it.

5.13 Related Work

Zaleski et al.[78] describe Yeti, a gradually extensible trace interpreter for Java. Yeti was designed so that JIT compilation could be added incrementally, a bytecode at a time. In order to improve the performance of code that it could only partly compile, Yeti needed to be able to interchange interpreted code and compiled code. This requirement is similar to that of HotPy for staged optimisation and results in aspects of the designs being similar. Yeti implements individual bytecodes, linear blocks (extended basic blocks) and traces all as callable functions, allowing them to be freely interchanged. Yeti constructs linear blocks, which are implemented using subroutine threading, from bytecodes on demand. It constructs traces from linear blocks when a back-edge becomes hot. Since Yeti implements Java, no specialisation is required other than the inlining of virtual calls, which happens as a side-effect of tracing.

Williams et al.[77] describe a specialising interpreter for Lua, which is not trace-based, but builds a specialised flow-graph for the executed program. It specialises on demand, but since Lua has only a few types, specialisation does not result in excessive duplication. Since Python has an unbounded number of types, this technique is not applicable directly to Python. Specialisation in HotPy is driven by trace selection. As far as I am aware, HotPy is the first VM which performs aggressive optimisation as bytecode-to-bytecode transformations.

Deferred Object Creation

Deferred object creation is a form of escape analysis. Escape analysis is usually used to allocate objects on the stack, rather than the heap, but that is not possible with the current GVMT garbage collector. The cache for storing non-deferred objects serves this role. Of course, not allocating an object at all is even cheaper than stack allocation.

Rigo[65] describes 'representation based specialization' in which changes to the representation of objects are made at runtime to reflect their usage. It main application is the unboxing of numbers, but is also applied to tuples and lists. The representation of a tuple of known size can be changed to a set of discrete values; this is in essence what deferred object creation is doing when a tuple is deferred. The main performance benefit of representation based specialization in Psyco was unboxing of integers, but the performance gains for floats and lists were considerable; see Section 2.5.2. Deferred object creation is a technique for implementing representation based specialization in the context of a stack-based bytecode interpreter.

5.14 Conclusion

In summary, the HotPy VM is designed to make full use of the abstract machine model outlined in Chapter 3 and the GVMT in particular. The restrictions on the design thus imposed have not been overly constraining.

The combination of these constraints, plus the delegation of garbage collection and JIT compilation issues to the toolkit, has helped to focus design decisions on the essential by removing the incidental.

Implementing the optimisations appropriate for a language like Python as a sequence of byecode-to-bytecode transformations works well. This method of implementing optimisations is conceptually straightforward, easy to implement and easy to debug. The ability to add a new bytecode with a few lines of code and have all the interpreters and the JIT compiler automatically updated makes experimentation extremely easy.

Chapter 6

Results and Evaluation

6.1 Introduction

Evaluating the effectiveness of a toolkit like the GVMT is difficult to do directly. To do so would require the development of two or more virtual machines from the same specification; one using the GVMT, and the others using different techniques. Even then it would be very hard to determine which characteristics of the resulting VMs were due to differences in the developers and which were due to the tools. Not only that, but the resources required would be well beyond those available for this research. Evaluating the usefulness of toolkits in general is an even more impractical task. Since a direct comparison is impractical, the usefulness of the GVMT must evaluated indirectly by comparing the VMs built using the GVMT with similar VMs constructed using other methods.

6.2 Utility of the GVMT and Toolkits in General

As discussed in Chapter 3, the task of building a VM for a dynamic language that integrates precise garbage collection and a JIT compiler is difficult without a toolkit. Of course, if building a VM for a dynamic language were just as difficult with the toolkit, then the developer might decide not to bother using the toolkit, perhaps opting to use a conservative garbage collector instead. Fortunately, toolkits do not need to impose excessive difficulties; the GVMT does not.

Consider the GVMT-Scheme VM discussed in Section 4.10. To create a Scheme interpreter with the same basic functionality would have required a similar amount of code, since both would be written in a mixture of C (for the core VM) and Scheme (for any libraries). The Scheme interpreter written without the toolkit would not have had to conform to the GVMT interface, but would have required integrating a conservative garbage collector. Integrating a conservative garbage

collector is a simple task, but so is conforming to the GVMT interface. Any optimisers would not have had the benefit of the consistency checking provided by the GVMT, and would thus have taken at least as long to develop. Overall, it seems reasonable to expect that without the toolkit, the resulting VM would be expected to have taken at least as much time to develop, and would lack precise garbage collection and JIT compilation.

6.3 Performance of the GVMT Scheme VM

Although the GVMT allows VMs to be developed quickly, in order to be useful it must produce VMs of adequate performance. The GVMT-Scheme VM was designed for clarity and speed of development, nonetheless it should provide good performance given that it performs basic optimisations, has precise garbage collection and uses JIT compilation.

6.3.1 Performance Comparison of Scheme VMs

Figure 6.1 compares the performance of GVMT-Scheme to three other implementations: Mzscheme, described in Section 2.6.9, Bigloo, described in Section 2.6.9, and SISC, a JVM based Scheme interpreter. SISC is a JVM based Scheme implementation. It claims to be the fastest Scheme implementation for the JVM, but does not perform any optimisations of the sort used in Mzscheme or Bigloo. Its complexity seems to be roughly on a par with that of GVMT-Scheme; the core of SISC contains rather more lines of code than GVMT-Scheme.

The three benchmarks are selected from the 'computer language benchmark game'[1]. All results are normalised to the Mzscheme interpreter without compilation (`mzscheme -i`). The '-i' suffix (`gvmt -i` and `mzscheme -i`) refers to the interpreter-only version (no compilation). Note the logarithmic scale.

As can be seen from Figure 6.1, GVMT-Scheme performance is comparable to that of Mzscheme. Considering the maturity of Mzscheme and given that GVMT Scheme was developed in under three weeks, this is a very satisfactory result which demonstrates the usefulness of the GVMT.

Both Mzscheme and GVMT-scheme outperform SISC by a large margin. The relatively poor performance of SISC, which runs on the JVM, serves to show the problems of adapting a language to a VM designed for a different type of language.

The performance of the Bigloo compiled code demonstrates that there is considerable room for performance improvement in the VMs. However, in order to

[1] `http://shootout.alioth.debian.org/`

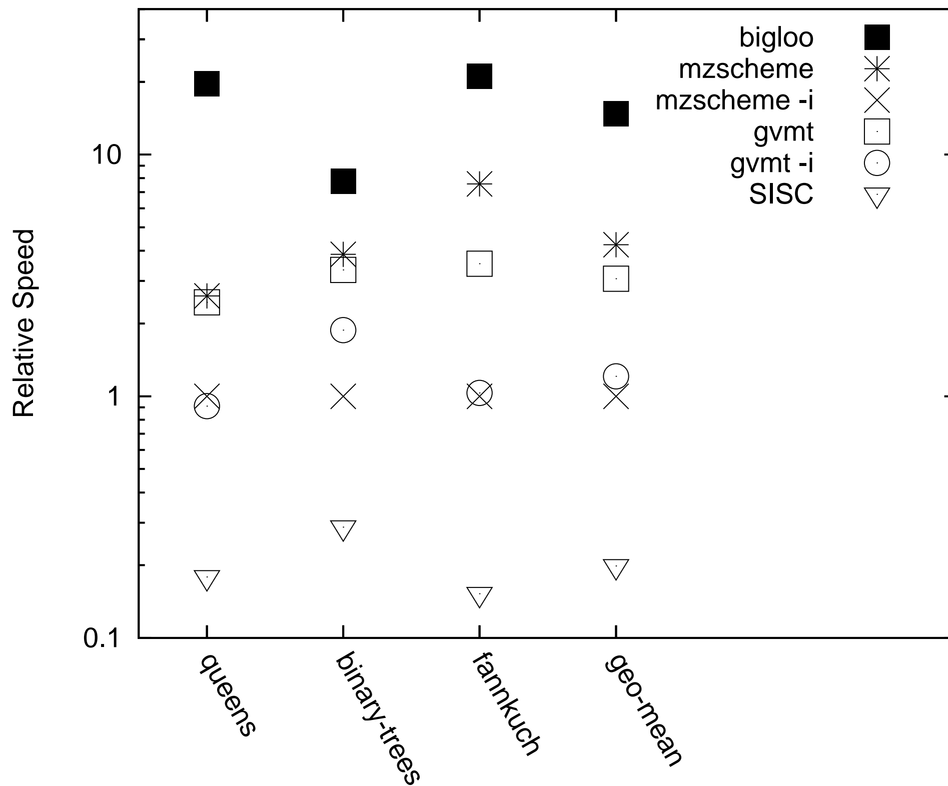


Figure 6.1: Performance of Scheme Implementations

move towards this level of performance, many sophisticated optimisations would be required.

6.4 Comparison of Unladen Swallow, the PyPy VM, and HotPy

In order to assess the quality of the HotPy VM, it will be compared with three other Python interpreters: the standard CPython interpreter, Unladen Swallow and the PyPy VM; see Sections 2.5.1, 2.5.5, 2.5.3 respectively.

Before comparing the performance of the virtual machines, a brief comparison of the differing designs of the four VMs is in order.

6.4.1 Relevant Design Details

These four different systems use different techniques to implement the VM. Both CPython and Unladen Swallow are built using the standard C and C++ compilers;

Unladen Swallow is a fork of CPython and uses LLVM to add JIT compilation. HotPy and PyPy (VM) are built using tools, the GVMT and PyPy (tool-chain), respectively.

HotPy and PyPy both have a generational garbage collector, whereas CPython and Unladen Swallow use reference-counting for garbage collection. Unladen Swallow performs profiling at runtime to guide subsequent compilation, whereas HotPy and PyPy use tracing to drive subsequent optimisation. HotPy performs most of its optimisations as bytecode-to-bytecode transformation. PyPy performs its optimisations on the same intermediate representation used to drive its custom machine-code generator. Unladen Swallow and HotPy both use LLVM for machine-code generation.

A Note on the Significance of Results

The aim of the benchmarking exercise here is to compare not the individual implementations, but the underlying techniques. Unfortunately it is very difficult to separate the two. Implementation details can account for a significant difference in performance. Consequently, when comparing differing implementations, it is probably wise not to attach much significance to small differences in performance.

The difference in performance between the base line performance of Unladen Swallow (which performs no optimisations) and Python 3 is about 10% (in Tables 6.1 and 6.2). This difference is wholly due to differing implementation details between Python2 and Python3. It thus seems reasonable to ignore such small differences and to regard larger differences, of say up to 30%, as of limited significance.

For example, when comparing HotPy to Unladen Swallow, the comparison is between the underlying methods of building the virtual machine, the differing approaches to optimisation *and* efficiency of the code in the implementation. Although it is possible to isolate these variables to some degree, it is only possible to be confident in a result if the differences are large.

When comparing the performance of two different settings of the same implementation, this caution does not apply.

6.4.2 Benchmarks Used

There is no standard benchmark suite for Python. The Unladen Swallow benchmark suite has become the de facto standard for benchmarking Python 2.x virtual machines, but has not been ported to Python 3, so could not be used. The 'pybench' suite that is included with Python is designed for benchmarking components of CPython and would give wildly varying results for a trace-based special-

ising optimiser; some benchmarks would be optimised to nothing, others might resist optimisation altogether. For example, one benchmark tests integer arithmetic by performing a number of simple operations on constants. HotPy (and PyPy) would optimise these away entirely.

Six programs were chosen as benchmarks. The benchmarks were chosen to test the VM rather than the supporting library. They exercise a range of the core features of the VM, namely integer and floating point arithmetic, list operations, generators, iterators, simple string manipulation and very basic I/O.

Two benchmarks, 'pystone' and 'richards', have been used for benchmarking Python since early versions. The 'gcbench' benchmark was taken from the Unladen Swallow benchmark suite, since a benchmark that stressed the garbage collector was required, and it is was trivial to port to Python 3. The remaining three benchmarks, 'fannkuch', 'fasta' and 'spectral-norm', were taken from the Computer Language Shootout Game. HotPy's limited library support ruled out a number of the Computer Language Shootout Game benchmarks, the remainder of the benchmarks tested either one library component, such as the regular expression engine or large integer arithmetic, or were floating point computations. Since Python is not generally used for computationally intensive tasks, including more than one floating point benchmark would bias the results.

The source code for all the benchmarks is in the `/benchmarks` subdirectory of the HotPy distribution.

6.4.3 Experimental Set Up

The machine used was an Intel Pentium 4 running at 3.00GHz with 1Mb of cache, running Linux. The machine was very lightly loaded (the X-server and Cron job scheduler were both turned off).

The versions of the VMs used were:

- HotPy — Revision 44 (built with GVMT revision 62), `http://code.google.com/p/hotpy/` `http://code.google.com/p/gvmt/`
- PyPy — Version 1.3, `http://pypy.org/download/pypy-1.3-linux.tar.bz2` and `http://pypy.org/download/pypy-1.3-linux-nojit.tar.bz2`
- Unladen Swallow — Revision 1159, `http://code.google.com/p/unladen-swallow/`
- CPython — Version 3.1.1, `http://www.python.org/ftp/python/3.1.1/Python-3.1.1.tgz`

Although HotPy has the potential to be multi-threaded, the experimental version was single-threaded only; compilation was done in the main interpreter thread. This seemed to be the fairest comparison as all the other VMs have a global interpreter lock. The GVMT garbage collector expects to run in a multi-threaded environment, so the garbage collector has to perform some synchronisation, even when running a single-threaded program. This seems to have no noticeable effect on performance.

Two variants of the HotPy VM were benchmarked. The two were the same except for the `getitem` and `setitem` methods for lists. The first version (marked 'C') has the `getitem` and `setitem` methods written in C. The second version (marked 'Py') has the methods written in Python. The Python implementations of the methods delegate to more specialised versions written in C. The different performance characteristics of the two libraries helps to illustrate the effect of optimisation on Python code.

All benchmarks were run on all virtual machines for three different durations, a short run of about a second ($\pm 60\%$) for the CPython implementation, a medium run of about ten seconds and a long run of about one hundred seconds. The short runs were used to demonstrate the lag effect of warm-up on the optimisers; the long runs were to allow the optimisers to warm up fully.

All benchmarks were run ten times, the slowest two discarded, and the rest averaged. The entries in the column labelled 'Mean' are the geometric means of the benchmark times.

All tables in this section show the performance relative to CPython; larger numbers are better. The configurations, inputs and full results, as times in seconds, for all runs are shown in Appendix F.

6.4.4 Base Line Performance

In order to assess the effectiveness of the toolkits in constructing simple, non-optimising VMs, the performance of the base interpreters was measured. All the VMs were run with JIT compilation turned off and, in the case of HotPy, all other optimisations were turned off as well. Tables 6.1 and 6.2 show the performance of the two HotPy variants, Unladen Swallow and PyPy relative to CPython. Table 6.1 shows the results for the short runs and Table 6.2 shows the results for the medium length runs. Results for the two lengths of runs are quite similar, which is unsurprising given that no runtime optimisations are taking place.

The small differences between the performance of Unladen Swallow (without compilation) and CPython are a consequence of Unladen Swallow being based on the 2.6 release of CPython, rather than the 3.1 release. The difference is small, but does show that implementation details do effect performance, even though the important features of the design are the same.

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
Un. Sw. (no JIT)	1.00	1.19	0.68	1.29	1.37	1.30	1.11
HotPy (base, C)	1.52	1.30	1.15	1.05	0.52	0.83	1.01
HotPy (base, Py)	1.50	1.00	1.13	0.33	0.36	0.83	0.74
PyPy (interpreter)	0.69	0.62	0.37	0.91	0.47	0.87	0.62

Table 6.1: Unoptimised Interpreters. Short Benchmarks. Speed Relative to CPython

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
Un. Sw. (no JIT)	0.99	1.18	0.66	1.28	1.35	1.35	1.10
HotPy (base, C)	1.54	1.34	1.20	1.11	0.50	0.88	1.03
HotPy (base, Py)	1.51	1.02	1.15	0.31	0.33	0.88	0.74
PyPy (interpreter)	0.66	0.60	0.35	0.87	0.44	0.91	0.60

Table 6.2: Unoptimised Interpreters. Medium Benchmarks. Speed Relative to CPython

HotPy(C) runs at about the same speed as CPython. HotPy(Py) and PyPy are both slower than CPython, by about the same margin. Of course, neither PyPy nor HotPy are designed to be run without any optimisation. The performance of HotPy(C) shows that a VM built with a toolkit need be no slower than one constructed conventionally, even without any attempt at optimisation. HotPy(Py) is noticeably slower than HotPy(C) as it must run extra Python code, which it is not optimising.

6.4.5 Full VM Performance

Tables 6.3 and 6.4 show the performance of the two HotPy variants and PyPy, with their default settings. Unladen Swallow is also tested with two different settings; the default setting, which compiles methods when hot, and with the JIT compiler always on.

HotPy(C) is fastest on the shortest benchmarks, by a tiny margin over HotPy(Py). PyPy is a little slower, but not by a significant amount. For the medium benchmarks, PyPy is the fastest by about 10%, not a significant margin.

The margins in the individual benchmarks are more significant. HotPy is faster for the gcbench and pystone benchmarks. The pystone benchmark is written in a procedural style and is mainly integer based. The use of tagged integers is probably a big help to HotPy for this benchmark. The gcbench benchmark is designed

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (JIT, C)	2.95	2.42	1.64	1.36	0.99	2.44	1.84
HotPy (JIT, Py)	2.94	2.38	1.56	1.46	0.94	2.45	1.82
PyPy (with JIT)	1.47	2.87	0.89	2.17	1.00	3.34	1.73
Un. Sw. (default)	1.07	0.48	0.37	0.68	0.84	1.06	0.70
Un. Sw. (always)	0.60	0.39	0.18	0.55	0.43	0.90	0.46

Table 6.3: Full VM. Short Benchmarks. Speed Relative to CPython

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
PyPy (with JIT)	3.82	7.23	3.93	4.15	1.09	11.74	4.23
HotPy (JIT, Py)	5.63	7.72	2.32	3.41	1.92	4.63	3.81
HotPy (JIT, C)	5.39	7.85	2.54	2.70	2.12	4.66	3.77
Un. Sw. (always)	1.05	0.92	0.49	1.56	1.25	1.67	1.08
Un. Sw. (default)	1.21	0.69	0.44	0.66	1.32	2.09	0.93

Table 6.4: Full VM. Medium Benchmarks. Speed Relative to CPython

to test garbage collection performance, but incidentally tests object initialisation performance as well. The fasta benchmark tests simple text formating and output. PyPy does not perform very well on this benchmark, only beating CPython by a small margin. The main loop in fasta is driven by a generator[2] so it is possible that the version 1.3 of PyPy does not optimise generators well. Although it starts more slowly PyPy is faster for the richards benchmark. The richards benchmark has a number of balanced `if` statements which can create a relatively large number of traces for the program size. PyPy is able to cope better with this thanks to its compiler, which is a lot faster than the LLVM-based compiler of HotPy. PyPy is clearly faster for the spectral-norm benchmark. The spectral-norm benchmark makes extensive use of floating point calculations, which is an area in which PyPy is particularly strong.

The performance of Unladen Swallow is surprisingly poor, starting very slowly and only just overtaking CPython for the medium length benchmarks. The richards benchmark seems to cause Unladen Swallow even more problems than the trace-based optimisers of HotPy and PyPy, which is surprising, as Unladen Swallow uses a function-at-a-time optimiser and shouldn't care when both branches of a conditional statement are taken.

In order to allow the slower LLVM-based compilers time to fully compile code, Table 6.5 shows relative performance for the long benchmarks. For the longest

[2]Generators in Python are a kind of iterator in form of a function that includes a `yield` expression. Each `yield` expression suspends execution of the function and returns a value. The generator is resumed by calling its `__next__` method.

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (JIT, Py)	9.77	12.86	4.14	5.09	2.64	7.24	6.08
HotPy (JIT, C)	8.82	13.54	4.24	3.64	2.96	7.26	5.83
PyPy (with JIT)	7.31	9.00	6.82	4.59	1.14	12.49	5.55
Un. Sw. (always)	1.09	1.09	0.60	1.89	1.61	1.83	1.26
Un. Sw. (default)	1.13	0.73	0.45	0.66	1.58	1.74	0.94

Table 6.5: Full VM. Long Benchmarks. Speed Relative to CPython

runs, HotPy outperforms PyPy by an insignificant margin. HotPy is noticeable faster than PyPy for integer work (pystone), and slower for floating point (spectral-norm). This suggests that HotPy would benefit from better optimisation of floating point computations. Conversely, PyPy would benefit from improved integer performance and better handling of generators (if that is the problem in the fasta benchmark). Unladen Swallow's performance is still relatively poor, but improved. Informal experiments showed that the performance of Unladen Swallow did not improve by much with even over longer runs.

6.4.6 Interpreter-Only Performance

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (int-opt, C)	3.98	3.10	2.44	1.67	0.81	2.17	2.11
HotPy (int-opt, Py)	3.97	3.08	2.23	1.50	0.79	2.17	2.03
PyPy (interpreter)	0.69	0.62	0.37	0.91	0.47	0.87	0.62

Table 6.6: Optimised Interpreters. Short Benchmarks. Speed Relative to CPython

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (int-opt, C)	4.30	3.20	2.59	1.71	0.77	2.34	2.19
HotPy (int-opt, Py)	4.30	3.18	2.38	1.71	0.75	2.34	2.14
PyPy (interpreter)	0.66	0.60	0.35	0.87	0.44	0.91	0.60

Table 6.7: Optimised Interpreters. Medium Benchmarks. Speed Relative to CPython

For some environments a JIT compiler is not available. Possibly the host device lacks sufficient memory, or the resources for porting the JIT compiler are not available. To simulate this case all the VMs are benchmarked with the JIT compiler disabled, but other optimisations left functioning. The results are shown in Tables 6.6 and 6.7. HotPy outperforms CPython by a factor of two, and outperforms

PyPy by a factor of three. This is an additional advantage of performing optimisations at the bytecode level; large performance gains can be made while keeping the advantages of an interpreter, namely portability and ease of maintenance.

It is worth pointing out that PyPy makes no attempt to optimise this case. It is probable that by applying some of the optimisations used in the compiler, and executing the resulting intermediate form, the PyPy interpreter could be made faster.

6.4.7 Comparing Compilation to Other Optimisations

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (int-opt, C)	3.98	3.10	2.44	1.67	0.81	2.17	2.11
HotPy (int-opt, Py)	3.97	3.08	2.23	1.50	0.79	2.17	2.03
Un. Sw. (default)	1.07	0.48	0.37	0.68	0.84	1.06	0.70
Un. Sw. (always)	0.60	0.39	0.18	0.55	0.43	0.90	0.46

Table 6.8: Interpreter vs. Compiler. Short Benchmarks. Speed Relative to CPython

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (int-opt, C)	4.30	3.20	2.59	1.71	0.77	2.34	2.19
HotPy (int-opt, Py)	4.30	3.18	2.38	1.71	0.75	2.34	2.14
Un. Sw. (always)	1.05	0.92	0.49	1.56	1.25	1.67	1.08
Un. Sw. (default)	1.21	0.69	0.44	0.66	1.32	2.09	0.93

Table 6.9: Interpreter vs. Compiler. Medium Benchmarks. Speed Relative to CPython

	gcbench	pystone	richards	fannkuch	fasta	spectral	Mean
HotPy (int-opt, C)	6.20	3.24	2.61	1.72	0.90	2.40	2.41
HotPy (int-opt, Py)	6.15	3.22	2.33	1.74	0.87	2.39	2.35
Un. Sw. (always)	1.09	1.09	0.60	1.89	1.61	1.83	1.26
Un. Sw. (default)	1.13	0.73	0.45	0.66	1.58	1.74	0.94

Table 6.10: Interpreter vs. Compiler. Long Benchmarks. Speed Relative to CPython

It is folklore that high performance in virtual machines is synonymous with JIT compilation. Whilst this is generally true for static languages, it is not neces-

sarily so for dynamic languages. Tables 6.8, 6.9 and 6.10 compare the performance of Unladen Swallow and HotPy in interpreter-only mode. Unsurprisingly for the short benchmarks HotPy is much faster. The difference is still large for the medium benchmarks.

For the longest benchmarks, Unladen Swallow with the JIT always on is faster than HotPy for two of the benchmarks. The HotPy interpreter is faster on the other long benchmarks, some by a large margin, and is significantly faster on average. On the default setting, Unladen Swallow speeds up the fasta and spectral norm benchmarks, but its overall performance is poor. Although HotPy appears to speed up on the gcbench benchmark from the medium to the long runs, this is in fact a slow down by CPython and Unladen Swallow. This slow down is probably caused by the garbage-cycle collector which has non-linear behaviour.

The relatively poor performance of Unladen Swallow adds weight to the argument that dynamic languages, such as Python, are just not amenable to the sort of optimisations used for static languages. Of course, once the dynamic form of the program has been transformed into a form that is more statically-typed, using tracing and specialisation, then compilation to machine code is a useful technique.

6.5 Aspects of Virtual Machine Performance

Although the goal of comparing the different virtual machines was to see the effects of differing construction techniques, it also shed some light on the relative value of differing optimisation techniques. This merits further examination. HotPy can be used as an experimental platform, as it is designed so that the various optimisations are modular and can be turned on or off independently. The interactions between various optimisations for dynamic languages can be explored by running HotPy with different settings.

The design of HotPy is such that all optimisations, including the compiler, work on traces. It is therefore impossible for HotPy to do any optimisations without first tracing. That is not to say that such optimisations cannot be done without tracing. Williams et al.[77] implement a specialising interpreter for Lua, in which specialisation is performed on demand. There is no separate tracing phase. They report speed-ups of about 30%. However, since Lua and Python are quite different, it is very hard to make any meaningful comparison of their results with the results for HotPy.

6.5.1 Permutations

Apart from tracing, all other optimisation passes can be turned on or off independently. As described in Section 5.5.1, the HotPy optimisers form a chain: tracing,

specialisation, deferred object creation (DOC), peephole optimisations and compilation. Since the optimisers are designed to work as a chain, each pass may not produce as clean code as it could, as each pass relies on the later passes to clean it up. As a consequence, all permutations are run with the peephole optimiser on, in order to minimise this effect.

The same set of benchmarks and durations, as described in Section 6.4.3, were used. The permutations of optimisations used were:

- No tracing; the base-line interpreter.
- Tracing only. (T)
- Tracing and specialising. (TS)
- Tracing and DOC. (TD)
- Tracing, specialising and DOC. (TSD)
- Tracing and compilation. (TC)
- Tracing, specialising and compilation. (TSC)
- Tracing, DOC and compilation. (TDC)
- Full, all four passes. (TSDC)

	T	TS	TD	TSD	TC	TSC	TDC	TSDC
Short Benchmarks	1.10	1.91	0.96	2.11	0.76	1.46	0.75	1.83
Medium Benchmarks	1.09	1.96	0.95	2.19	1.05	2.51	1.04	3.78
Long Benchmarks	1.16	2.14	1.01	2.41	1.24	3.42	1.25	5.83

Table 6.11: HotPy(C) Performance Permutations. Speeds Relative to CPython

	T	TS	TD	TSD	TC	TSC	TDC	TSDC
Short Benchmarks	0.80	1.30	0.73	1.97	0.54	1.07	0.56	1.82
Medium Benchmarks	0.78	1.31	0.71	2.14	0.74	1.70	0.78	3.80
Long Benchmarks	0.83	1.41	0.76	2.34	0.89	2.32	0.93	6.11

Table 6.12: HotPy(Py) Performance Permutations. Speeds Relative to CPython

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	1.73	—	2.18	—	1.92	—	2.45
Medium Benchmarks	1.80	—	2.31	—	2.39	—	3.63
Long Benchmarks	1.84	—	2.38	—	2.75	—	4.67

Table 6.13: Speed Up Due to Adding Specialiser; HotPy(C).

Tables 6.11 and 6.12 show the mean speeds of the various permutations relative to CPython. Results for the individual benchmarks are shown in Appendix F.

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	0.87	1.10	—	—	0.98	1.25	—
Medium Benchmarks	0.87	1.12	—	—	1.00	1.51	—
Long Benchmarks	0.87	1.12	—	—	1.00	1.71	—

Table 6.14: Speed Up Due to Adding D.O.C.; HotPy(C).

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	0.69	0.77	0.77	0.87	—	—	—
Medium Benchmarks	0.96	1.28	1.10	1.73	—	—	—
Long Benchmarks	1.07	1.60	1.24	2.42	—	—	—

Table 6.15: Speed Up Due to Adding Compiler; HotPy(C).

The interrelations between the passes are shown more clearly by Tables 6.13, 6.14 and 6.15 for HotPy(C) and by Tables 6.16, 6.17 and 6.18 for HotPy(Py). The tables show the relative speed-ups for individual passes, for the mean of the benchmarks. Each column shows the speed-ups for adding the optimisation pass for that table, to the permutation of that column.

It is immediately clear that specialisation is important for performance. The gains for specialisation by itself are large. Not only that, specialisation significantly improves the quality of input to the other optimisers, generating even larger gains. It is worth noting that both of the specialisation-without-compilation settings (TS and TSD) outperform both of compilation-without-specialisation settings (TC and TDC) for all the benchmarks of any duration.

The utility of deferred object creation depends a lot on which other optimisations are used. It is useful when combined with specialisation and even more useful when compilation is used as well. When used with neither specialisation nor compilation (TD), it actually slows code down. This is to be expected since DOC relies on precise type information to avoid having to create objects across calls and operators. The interaction with compilation is a result of DOC generating more bytecodes, that perform slightly less work, when no type information is

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	1.63	—	2.70	—	1.98	—	3.25
Medium Benchmarks	1.68	—	3.02	—	2.29	—	4.89
Long Benchmarks	1.71	—	3.09	—	2.62	—	6.56

Table 6.16: Speed Up Due to Adding Specialiser; HotPy(Py).

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	0.92	1.52	—	—	1.03	1.70	—
Medium Benchmarks	0.91	1.63	—	—	1.04	2.23	—
Long Benchmarks	0.92	1.66	—	—	1.05	2.63	—

Table 6.17: Speed Up Due to Adding D.O.C.; HotPy(Py).

	T	TS	TD	TSD	TC	TSC	TDC
Short Benchmarks	0.68	0.82	0.76	0.92	—	—	—
Medium Benchmarks	0.95	1.30	1.09	1.78	—	—	—
Long Benchmarks	1.07	1.64	1.23	2.61	—	—	—

Table 6.18: Speed Up Due to Adding Compiler; HotPy(Py).

available. This results in code that is a little faster once compiled, but is slower when interpreted. DOC is a worthwhile optimisation, since when paired with specialisation it always results in speedups; in the best cases it more than doubles performance.

Compilation by itself is of no use as it tends to slow code down, but is very useful when following on from other optimisations. Compilation is doubly reliant on the quality of code generated by the upstream passes. Not only can the compiler produce better machine code from better bytecode, it can do more quickly, allowing more code to be compiled which further increases performance.

Specialisation Is Key

The results clearly show that trace-driven specialisation is the *key* optimisation for HotPy, and by implication for the optimisation of other dynamic languages. That specialisation is important for optimising dynamic languages is not surprising; what is slightly surprising is its effect on other optimisations. Without specialisation, the D.O.C pass is essentially useless and compilation is not much better. Compilation is at least seven times as effective (measured in terms of speedup) with specialisation than without.

Specialisation unlocks the other optimisations. Although the speed up from DOC is about the same as that from specialisation and the speed up from compilation exceeds these, the other optimisations only work well with specialised input.

The poor performance of compilation without the help of specialisation may shed some light on the performance of Unladen Swallow. Unladen Swallow does some profiling to gather type-information at runtime, but without trace-driven specialisation this appears to be of limited use.

6.6 Memory Usage

Increased performance often comes at the cost of increased memory usage, as time-space trade offs can be made. Optimisers, especially compilers can use considerable amounts of memory.

6.6.1 Experimental Method

Real memory usage is difficult to measure with an operating system that supports virtual memory, since the real memory available to a process is effectively hidden by the operating system. Linux, which was the system used for development and measurement, provides no consistent measure of real memory usage. Although it is impossible to measure real memory usage without modifying the operating system, it is possible to measure the minimum amount of virtual memory that a VM needs to complete a benchmark.

Each benchmark (long version) was run repeatedly on each VM, successively increasing the amount of the maximum amount of memory available to the process, using the linux `ulimit -v` utility, until the process completed properly, 5 times in a row.

6.6.2 Results

Table 6.19 shows the minimum amount of memory (in megabytes) required to run each benchmark; smaller number are better. PyPy without a JIT is not considered as its performance is worse than CPython's. The 'hello' benchmark is a single line benchmark to test how much memory each VM requires in order to start up and shut down.

	hello	gcbench	pystone	richards	fannkuch	fasta	spectral
CPython	6	97	7	7	6	7	7
Un. Sw. (default)	16	111	21	22	20	21	18
HotPy (full)	44	113	63	65	64	62	62
HotPy (no comp)	26	87	27	28	29	27	27
PyPy (with JIT)	37	92	40	41	40	48	40

Table 6.19: Long Benchmarks. Minimum Required Virtual Memory

As can be seen CPython uses considerably less memory than any of the other VMs, except for the GCBench benchmark where HotPy (interpreter only) and PyPy use a little less than CPython.

Considering all but the GCBench benchmarks, Unladen Swallow uses 11-15 Mbytes more than CPython, PyPy uses 33 to 41 Mbytes more, HotPy (without the compiler) uses 17 to 20 Mbytes more and HotPy (full) uses 50 to 55 Mbytes more.

Memory usage can be broken into two parts; fixed overheads and dynamic memory use. Clearly HotPy and PyPy have large fixed memory overheads. The fixed overhead of HotPy (with compiler) is particularly large.

Fixed Memory Overhead of HotPy

The fixed memory overheads of HotPy can be broken down into three parts; translation overheads, memory management overhead and the JIT compiler. These are mainly attributable to the GVMT, rather than HotPy itself.

HotPy (without the compiler) uses 20 to 23 Mbytes more than CPython (except for GCBench). Running HotPy with a memory debugger shows no significant memory leaks. The GVMT runtime allocates an 8 Mbyte nursery at start up. Recompiling GVMT to use a 1 Mbyte nursery reduces the memory usage by up to 8 Mbytes. However, with a 1 Mbyte nursery GCBench uses almost as much memory and runs quite a lot slower; a variable sized nursery is obviously required. The GVMT linker also lays out memory rather sparsely, taking 1.5 Mbytes for data that could be fitted in 0.5 MBytes. In total, the heap is about 8 Mbytes larger than it needs to be at start up.

By default, Linux allocates 2 Mbytes of stack space per thread. GVMT creates a collector thread and a finaliser thread in addition to the main thread. The separation of the HotPy VM frame stack from the underlying GVMT stack means that HotPy can run deeply recursive programs with very little C stack. This means that the stack space for each thread can be reduced to 100 Kbytes or less. Experimentally reducing the stack space to 100 Kbytes (using `ulimit`) reduces memory usage by over 5 MBytes.

Clearly most of the overhead is an artifact of the implementation, rather than a fundamental issue. Removing the combined overheads of nursery, layout and stacks would reduce the fixed memory overhead from 20 MBytes down to 6 or 7 Mbytes. This should be addressed in future versions of the GVMT and HotPy.

The HotPy compiler is built as a separate dynamically linked library, and adds 18 Mbytes for the 'hello world' program which loads the compiler, but does not run it. This compares unfavourably to Unladen Swallow which adds about 10 Mbytes fixed overhead to CPython.

Dynamic Memory Usage of HotPy

The dynamic overhead of HotPy, that is the extra memory required to run is dominated by the heap memory required for objects and the temporary memory required by the LLVM compiler backend.

Both HotPy and PyPy are able to reduce the memory footprint of object dictionaries by sharing the keys. The effect of this is shown in the GCBench results. CPython and Unladen Swallow require about 90 Mbytes more than the other benchmarks. HotPy requires about 60 Mbytes and PyPy requires about 50 Mbytes. Although HotPy uses more memory than PyPy for its heap objects, it uses a simpler approach than PyPy and uses a lot less memory than CPython.

The HotPy compiler uses a further 16 to 20 MBytes when executing. This is considerably more than Unladen Swallow which adds up to 5 Mbytes more when running. The reasons why the HotPy compiler uses so much more memory than Unladen Swallow are not clear. Both require LLVM and the GVMT generated part of the compiler is less than 1 Mbyte. The final machine code by LLVM should be compact and efficient; LLVM is competitive with GCC and the JIT compiler generates the same code as the offline version. The machine code generated by the HotPy VM seems to be efficient, it outperforms Unladen Swallow by a considerable margin. It is possible that the LLVM intermediate representation generated by the GVMT compiler is large and for some reason causes LLVM to use considerable memory to perform its optimisations; the GVMT uses optimisations in LLVM equivalent to the -O2 setting for the static compiler.

The best way to reduce dynamic memory use would be to replace LLVM with a leaner compiler.

6.7 Effect of Garbage Collection

gcbench	pystone	richards	fannkuch	n-body	richards
40.5%	6.4%	4.5%	6.2%	2.9%	5.8%

Table 6.20: CPython GC percentages

Non-GC Speed up	gcbench	pystone	richards	fannkuch	n-body	richards	Mean
× 2	1.4	1.9	1.9	1.9	1.9	1.9	1.8
× 3	1.7	2.7	2.8	2.7	2.8	2.7	2.5
× 5	1.9	4.0	4.2	4.0	4.5	4.1	3.6
× 8	2.1	5.5	6.1	5.6	6.7	5.7	5.0

Table 6.21: Theoretical CPython Speedups

Table 6.20 shows the percent time spent in explicit memory management function in CPython for the medium benchmarks. The data was gathered using the oprofile profiling tool and summing the execution time of all functions explicitly involved in allocation or deallocation. No functions which initialise objects were included, nor was any attempt made to measure the overhead of reference counting.

Table 6.21 show the expected overall speed-up of the VM if all other components of the VM were sped up by the factor on the left, but no attempt made to improve garbage collection performance.

Obviously this is an over-simplification, but it suggest that reference counting does not prevent useful improvements in performance. However, if large speed-ups are required then the overhead of poor garbage collection will become a problem. To achieve a speed-up of five, the stated goal of Unladen Swallow and an achievable goal, as PyPy and HotPy have demonstrated, would require speeding up the remainder of the VM by a factor of eight; quite an ambitious target.

6.8 Potential for Further Optimisation

Although PyPy and HotPy achieve significant speedups over CPython, they remain slow compared to VMs for Java or C#, let alone compiled C or Fortran.

Although it is impossible to put a definite upper bound on the performance of a Python VM, it is reasonable to assume that a Python VM is not going to be as fast as compiled C code or Java running on a modern VM. A direct comparison of HotPy and PyPy to compiled C and a modern Java VM is not necessary meaningful due to many dynamic features of Python that are not present in statically typed languages. Nonetheless a comparison does have some value. It provides a (rather high) upper bound on expectations for possible performance improvements, and gives some objective way of measuring the quality of optimisation.

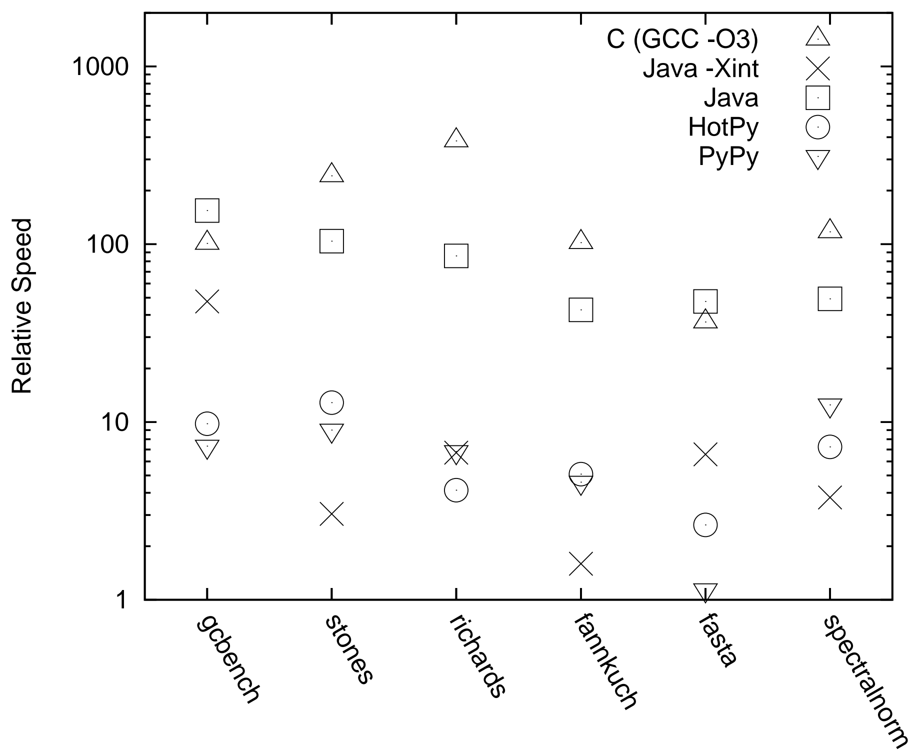


Figure 6.2: Performance of HotPy and PyPy compared to C and Java

Figure 6.2 shows the performance of HotPy, PyPy compared with C and Java equivalents of the Python benchmarks. The C and Java versions of the first three benchmarks are broadly similar to the Python versions. The C version of GCBench is a direct translation of the Java version, using the Boehm conservative collector to perform memory management. The second three benchmarks are taken from the Computer Language Benchmark Game, and are more idiomatic for all three languages.

Source code is included in the `benchmarks` folder of the HotPy distribution. The Java VM used was OpenJDK 1.6.0_0 (build 1.6.0_0-b11, mixed-mode, sharing), using both the default setting and the interpreter-only setting (-Xint). The C compiler was GCC 4.2.4 using -O3 optimisation.

It is clear from Figure 6.2 that there is plenty of scope for improving performance. How much HotPy or PyPy could be improved is far from clear. What is clear is that minor efficiency improvements, such as better machine code generation or lower memory management overhead is not going to make Python as fast as Java; completely new optimisations are required.

6.8.1 Quality of Optimisation

Given some baseline performance and a target optimisation it is possible to calculate a quality metric for an optimisation. For a baseline time, t_b a target time t_t, and the time for a VM being assessed t_v, a metric can be calculated to assess the 'quality' of the optimisations applied. The metric is designed so that no speedup gives a metric of 0 and achieving the target gives a metric of 1.

A logarithmic, rather than a linear, metric is chosen. The logarithmic metric $(log(t_b) - log(t_v))/(log(t_b) - log(t_t))$ gives results on a range of 0 to 1 and gives a metric of 0.5 when the speedup for the VM is the square root of the target speedup. The linear metric, $(t_b/t_v - 1)/(t_b/t_t - 1))$, would give unduly small values for significant speedups in cases where t_t is much smaller than t_b . Figure 6.3 shows the 'quality', using the logarithmic metric, of the optimisations used in HotPy and PyPy measured against CPython as the baseline and Java (OpenJDK) as the target.

With the exception of the fasta benchmark, the qualities of HotPy and PyPy cluster around 0.5, a sort of half way mark. The fasta benchmark is the odd one out; the quality metric for HotPy is low and for PyPy is close to zero. Although, the fasta benchmark has been optimised especially for CPython, its style is not that unusual, making heavy use of generators and list comprehensions. There is no compelling reason why this should not be optimised as well as the other benchmarks. This merits further investigation, perhaps suggesting that generators and list comprehensions are harder to optimise than other constructs.

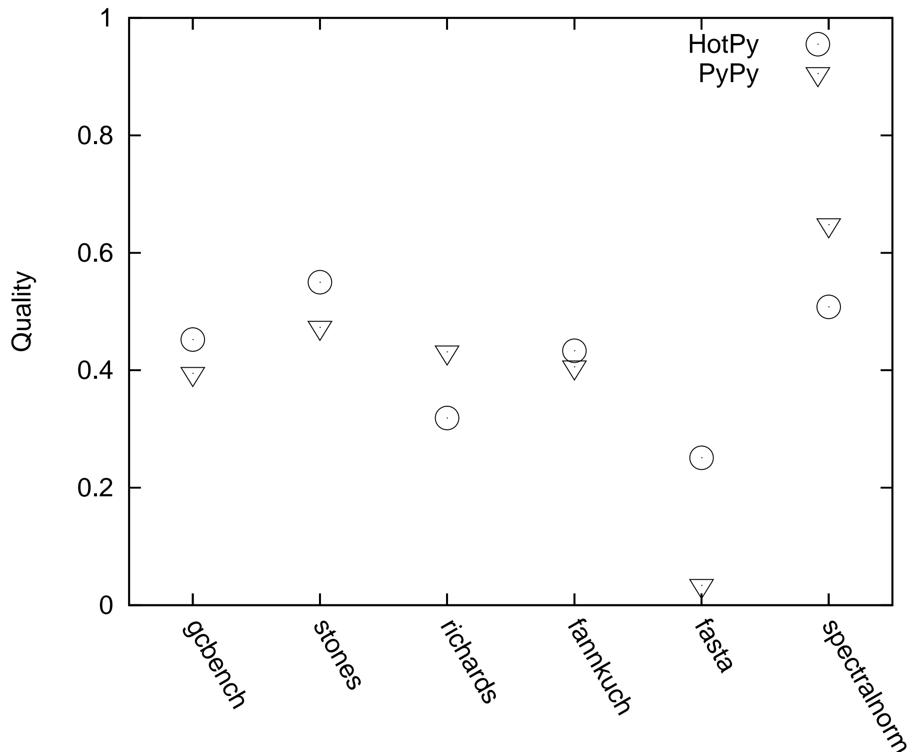


Figure 6.3: Quality of HotPy and PyPy Optimisations Measured against Java (OpenJDK)

6.9 Conclusions

The results in Sections 6.4.7 and 6.5.1 show that although JIT compilation is necessary for high performance, it is not sufficient. Without applying optimisations suitable for dynamic languages before generating machine code, the resulting machine code will likely be bulky and inefficient.

The effective optimisation of dynamic languages requires a number of complementary optimisations. Although tracing and specialising can yield worthwhile performance gains, to achieve larger gains requires a combination of optimisations including JIT compilation. A high-performance garbage collector is also required or the time spent managing memory will dilute the hard won performance gains.

Although the optimisations are complementary it is clear that specialisation is the key optimisation; without it all other optimisations are of little or no value.

Analysis of the memory usage of HotPy shows that the GVMT needs some refinements of its garbage collector implementation, in order to reduce wasted space. Replacing LLVM would also reduce memory usage.

Although HotPy and PyPy have similar mean performance, the differences in-

dicate that each VM has areas which could implemented better, yielding further performance improvements without novel optimisations. The comparison with C and Java shows even with refinements, neither HotPy nor PyPy achieve anywhere near the performance of statically typed, compiled code. How far this gap can be closed remains an open question. The results in Section 6.8.1 show a measure of the quality of optimisation, but there no way to determine what level of quality is attainable.

Chapter 7

Conclusions

7.1 Review of the Thesis

In the introduction, the central thesis of the dissertation was stated as:

> The best way, in fact the only practical way, to build a high-performance virtual machine for a dynamic language is using a tool or toolkit.
>
> Such a toolkit should be designed around an abstract machine model.
>
> Such a toolkit can be constructed in a modular fashion, allowing each tool or component to use pre-existing tools or components.
>
> Using such a toolkit, it is possible to build a virtual machine that is at least as fast as virtual machines built using alternative techniques, and to do so with less effort.

The enormous resources put into the JVM and CLR indicates that creating a virtual machine that combines precise GC and JIT compilation is no easy task. Of the many VMs discussed in Chapter 2, very few managed to combine precise GC and JIT compilation, and those that did were for languages simpler than Python. It is reasonable to conclude that integrating the complex features of a VM requires some sort of tool support.

It is unlikely that using a toolkit based around an abstract machine is the only way to construct a VM for dynamic languages, but there are no compelling alternatives. As argued in Chapter 3, using a toolkit allows clear separation of low-level and high-level concerns. Designing the toolkit around a well-defined abstract machine brings considerable benefits in modularity. The GVMT, discussed in Chapter 4, demonstrated that toolkit can be made by modifying and wrapping pre-existing tools in a modular fashion.

The utility of the GVMT was demonstrated by the construction of two different VMs. In Chapter 6 comparison of the VMs constructed by the GVMT showed that VMs constructed by toolkits can perform at least as well as those constructed by other means.

What is not clear is whether a tool with the power and generality of the GVMT is required. For the construction of a single virtual machine a simpler special purpose tool might be more appropriate. Nonetheless, given that the GVMT does exist, it is a valuable tool for experimentation with VM design.

7.2 Significant Results

As well as demonstrating that using a toolkit is an effective way to create virtual machines, this dissertation also illuminates other aspects of the construction of virtual machines for dynamic languages.

7.2.1 Bytecode-to-Bytecode Optimisations

The most important result is showing the effectiveness of bytecode-to-bytecode translations as a means of optimising execution traces. Large speed ups are possible in purely interpreted code, by tracing and then applying specialisation and escape analysis to the resulting traces. The speed ups gained are complementary to those from compilation.

7.2.2 Comparison of Optimisation Techniques

Analysis of the HotPy VM shows the relative power of various optimisation techniques and the dependencies between those optimisations. Specialisation was shown, in addition to providing a speedup of its own, to be key to the other optimisations. Essentially, without specialisation, the other optimisations are not worthwhile.

7.3 Dissertation Summary

As discussed in the introduction, the implementation of a high-performance virtual machine for dynamic languages is a hard task. It is the central thesis of this dissertation that the implementation of these VMs becomes more manageable by using a toolkit, without impairing performance. The use of a toolkit allows proper

separation of high-level and low-level concerns. Low-level concerns such as integration of the garbage collector and machine-code generation are managed by the toolkit, which leaves the developer better able to address the high-level issues.

The abstract machine model (Chapter 3) provides this same separation of concerns within the toolkit. The front-end tools target the abstract machine, and the back-end tools need no knowledge of how the abstract machine code was generated. This is much like a re-targetable compiler.

The generality of the toolkit makes it flexible. When implementing the Glasgow Virtual Machine Toolkit (GVMT) (Chapter 4) I implicitly assumed that VM function calls would map to GVMT function calls, and that the JIT compiler would be compiling whole functions. However, HotPy ended up using trace-based optimisation. Compiling traces was no problem as the JIT compiler can compile arbitrary (terminated) sequences of bytecodes, and was able to compile traces just as well as functions.

The implementation of HotPy, as described in Chapter 5, makes full use of the GVMT provided capabilities. The facilities for exception handling, JIT compilation and garbage collection are used to the full. By using the GVMT, the implementation of HotPy has no dependence on the low-level implementation details of GVMT provided components. As the implementer of HotPy, I was unaware of what numerical value was assigned to each opcode, when the garbage collector was run, or how the garbage collector found all references. The JIT compiler was always available. Whenever a new bytecode was added or an old one removed, the JIT compiler was automatically updated; the interpreter and JIT compiler always obey the same semantics. The toolkit also ensured that all the bytecode processors conformed to the same bytecode format.

The only real restrictions that the GVMT puts on the VM developer are that the input to the JIT compiler must be bytecodes, and the necessary limitations on the use of heap pointers. The requirement that the input to the JIT compiler must be bytecodes forces the developer to implement optimisations as bytecode-to-bytecode transformations. As argued in Chapter 3, this is not a problem as bytecode is a good intermediate representation. The HotPy optimisers described in Section 5.5 were easy to implement and debug. The support for secondary bytecode interpreters provided by the GVMT made them easy to implement. They were easy to debug as the output could be disassembled and visually scanned, which made errors easy to locate.

The results shown in Chapter 6 clearly demonstrate that the enforced separation of high-level and low-level optimisation is not harmful to performance. Not only does separating the optimisations not harm performance, it allows the optimisations to be used independently. This was most clearly shown in Section 6.4.6, where disabling compilation allowed HotPy to still perform reasonably well, but crippled the other optimising VMs. The ability to separate high-level optimisations from low-level ones allows the relative utility of these to be demonstated in

Section 6.5.

The effectiveness of interpreter-only optimisations is a key discovery of this research. As shown in Section 6.4.7, which compares an interpreter-only optimising VM (HotPy) with a compiler-based VM (Unladen Swallow), trace-based bytecode-to-bytecode optimisations can be an effective way of optimising dynamic languages. Not only is trace-based bytecode-to-bytecode optimisation an effective optimisation, it is an ideal precursor to conventional JIT compilation.

Although compilation to machine code is still valuable, it should be implemented *after* other optimisations. It is not only Python to which these arguments apply. The performance of Javascript VMs is important to many web-based applications; Javascript programs are often short and the cost of JIT compilation, unless carefully engineered, can outweigh the advantages. Bytecode-to-bytecode translation provides a possible alternative to JIT compilation as it will, in general, be faster and use less memory.

Evaluating the memory usage of HotPy shows that the GVMT in its current form creates VMs with large memory footprints. However, analysis shows that this problem is not fundamentally due to the use a toolkit, rather it is an artifact of implementation.

Finally, the performance of HotPy and PyPy were compared to compiled C and Java (the OpenJDK VM). Although both VMs manage achieve large speed ups relative to CPython, their performance is much worse than either compiled C or the Java VM. The performance of highly dynamic languages can still be improved by a considerable degree. How that should be done has yet to be discovered.

7.4 Future Work

Further research can be divided into performance enhancements and the evaluation of different VM optimisations. The lessons learnt can also be applied to existing VMs.

The maximum benefit from bytecode-to-bytecode optimisations might be achieved, not by large improvements in research VMs, but by applying these optimisations to the mainstream VMs. Several dynamic languages, particularly Ruby, would benefit from implementing trace-based bytecode-to-bytecode optimisations. However, this dissertation focuses on Python. The CPython VM could be improved by applying the results of Chapter 6.

7.4.1 Applying the Research to CPython

My recommendations for improving the performance of the CPython VM are therefore as follows:

1. Determine a strategy for improving the garbage collector. This strategy should be formulated first so that subsequent optimisations do not prevent it being implemented.
2. Implement a trace recorder for recording traces and a super-interpreter for managing the execution of traces. The resulting traces, and output from all subsequent optimisers, should be executable; this will allow separate development and testing.
3. Implement a specialisation pass, to specialise traces.
4. Implement a deferred object creation (DOC) pass, and peephole optimiser.
5. Once the specialisation and DOC passes are stabilised and the bytecode format is fixed, then a JIT compiler can be implemented. Since the input to the JIT is already well optimised, a direct translation to LLVM IR[1], or equivalent, should work well.
6. Implement the strategy, determined in the first step, for improving the garbage collector.

The strategy for improving the garbage collector is outside the scope of thesis.

7.4.2 Performance Enhancements to the GVMT and HotPy

Performance enhancements for the GVMT are likely to be incremental changes of limited scope and of little interest to the academic community. The only potential for significant improvement is in the compiler implementation, which is rather slow. HotPy is a much more promising direction, as there is the potential for large and interesting performance improvements.

An Almost-Trace Compiler

Trace-based optimisations are important in VMs, not only to allow specialisation, but because trace-based JIT compilers can be much faster than conventional compilers yet still generate code of the same quality. The HotPy VM and potentially other research VMs built using the GVMT, use tracing at the bytecode level.

[1] `http://llvm.org/docs/LangRef.html`

However, these traces may not be proper traces at the abstract machine level, even though they are at the bytecode level.

For a system like HotPy, it would be good for the GVMT-generated compiler to be as fast as a trace-based compiler, and be able to compile the 'almost' traces that may result from proper traces at the bytecode level. If code quality does not matter, it is easy to make a faster compiler than the current LLVM compiler. The challenge would be to extend a trace-based compiler to handle 'almost' traces, producing quality code, but faster than the current LLVM-based compiler.

Other Bytecode-to-Bytecode Translations

HotPy, although considerably faster than CPython, still lags behind other language implementations. For example, the LuaJIT VM for Lua is much faster. Obviously, improving the performance of the underlying toolkit will help to reduce this difference, but there is still much room for improvement at the bytecode level. A first step would be to extend the DOC pass to be able to defer object creation across backward jumps at the end of loops and to unbox floats (and possibly complex numbers).

7.4.3 Evaluation of VM Optimisation Techniques

One potential use of the GVMT, and of HotPy, is as a fixed base for comparative evaluation of optimisation techniques. For example, a more precise examination of the relative merits of whole-function optimisation versus trace-based optimisation could be made by implementing both of these optimisations in a single VM built using the GVMT. The ability to reduce external factors to a minimum is necessary to perform truly meaningful comparisons. As the GVMT Scheme VM demonstrates, VMs can be constructed in a time frame that makes this sort of experimentation viable.

7.5 In Closing

The core message of this dissertation is that building a VM for a complex and evolving language like Python is much easier with a set of appropriate tools. The key reason for this is that a VM consists of a number of closely interacting parts that interface in ways that conventional programming languages do not support well. By converting the source code for the interpreter and libraries into abstract machine code, it is possible to analyse and transform this code. This enables the code generators to weave the garbage collector into the rest of the VM, and makes it possible to generate an interpreter and JIT compiler from the same source code.

The ability to change the interpreter source and have a new VM with a JIT compiler up and running within a minute or two is enormously helpful. The speed of development of known optimisations in the VM is increased considerably, and the ability to experiment very quickly helps with the design of new optimisations.

Appendix A

The GVMT Abstract Machine Instruction Set

Introduction

This appendix lists all 367 instructions of the GVMT abstract machine instruction set. The instruction set is not as large as it first appears. Many of these are multiple versions of the form OP_X where X can be any or all of the twelve different types. These types are I1, I2, I4, I8, U1, U2, U4, U8, F4, F8, P, R.

IX, UX and FX refer to a signed integer, unsigned integer and floating point real of size (in bytes) X. P is a pointer and R is a reference. P pointers cannot point into the GC heap. R references are pointers that can *only* point into the GC heap.

For all instructions where the type is a pointer sized integer, I4 and U4 for 32-bit machines or I8 and U8 for 64-bit machines, there is an alias for each instruction of the form OP_IPTR or OP_UPTR. E.g. on a 32-bit machine the instruction ADD_I4 has an alias ADD_IPTR.

TOS is an abbreviation for top-of-stack and NOS is an abbreviation for next-on-stack.

Each instruction is listed below in the form:

Name (inputs ⇒ outputs)

Instruction stream effect
Description of the instruction

#+ (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, adds them and pushes the result back to the stream.

#- (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, subtracts them and pushes the result back to the stream.

#n (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Push 1 byte value to the front of the instruction stream.

#2@ (— ⇒ operand)

2 operand bytes.
Fetches the next 2 bytes from the instruction stream. Combine into an integer, first byte is most significant.Push onto the data stack.

#4@ (— ⇒ operand)

4 operand bytes.
Fetches the next 4 bytes from the instruction stream. Combine into an integer, first byte is most significant.Push onto the data stack.

#@ (— ⇒ operand)

1 operand byte.
Fetches the next byte from the instruction stream. Push onto the data stack.

#[n] (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Only valid in an interpreter defintion. Peeks into the instruction stream and pushes the nth byte in the stream to the front of the instruction stream.

ADDR(name) (— ⇒ address)

Pushes the address of the global variable name to the stack (as a pointer).

ADD_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point add.

result := op1 + op2.

ADD_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point add.

result := op1 + op2.

ADD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer add.

result := op1 + op2.

ADD_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer add.

result := op1 + op2.

ADD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer add.

result := op1 + op2.

ADD_P (op1, op2 ⇒ result)

Binary operation: pointer add.

result := op1 + op2.

ADD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer add.

result := op1 + op2.

ADD_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer add.

result := op1 + op2.

ADD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer add.

result := op1 + op2.

ALLOCA_F4 (n ⇒ ptr)

Allocates space for n 32 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_F8 (n ⇒ ptr)

Allocates space for n 64 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I1 (n ⇒ ptr)

Allocates space for n 8 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I2 (n ⇒ ptr)

Allocates space for n 16 bit signed integers in the current control stack frame, leaving pointer to allocated

space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I4 (n ⇒ ptr)

Allocates space for n 32 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I8 (n ⇒ ptr)

Allocates space for n 64 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I4 (n ⇒ ptr)

Allocates space for n 32 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_P (n ⇒ ptr)

Allocates space for n pointers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_R (n ⇒ ptr)

Allocates space for n references in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction. ALLOCA_R cannot be used after the first HOP, BRANCH, TARGET, JUMP or FAR_JUMP instruction.

ALLOCA_U1 (n ⇒ ptr)

Allocates space for n 8 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U2 (n ⇒ ptr)

Allocates space for n 16 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U4 (n ⇒ ptr)

Allocates space for n 32 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U8 (n ⇒ ptr)

Allocates space for n 64 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U4 (n ⇒ ptr)

Allocates space for n 32 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

AND_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise and.

result := op1 & op2.

AND_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer bitwise and.

result := op1 & op2.

AND_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise and.

result := op1 & op2.

AND_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise and.

result := op1 & op2.

AND_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer bitwise and.

result := op1 & op2.

AND_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise and.

result := op1 & op2.

BRANCH_F(n) (cond ⇒ —)

Branch if TOS is zero to Target(n). TOS must be an integer.

BRANCH_T(n) (cond ⇒ —)

Branch if TOS is non-zero to Target(n). TOS must be an integer.

CALL_F4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit floating point.

CALL_F8 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 64 bit floating point.

CALL_I4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit signed integer.

CALL_I8 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 64 bit signed integer.

CALL_I4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit signed integer.

CALL_P (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a pointer.

CALL_R (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a reference.

CALL_U4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit unsigned integer.

CALL_U8 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 64 bit unsigned integer.

CALL_U4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit unsigned integer.

CALL_V (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return void.

D2F (val ⇒ result)

Converts 64 bit floating point to 32 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

D2I (val ⇒ result)

Converts 64 bit floating point to 32 bit signed integer. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

D2L (val ⇒ result)

Converts 64 bit floating point to 64 bit signed integer. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

DIV_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point divide.

result := op1 / op2. Rounds towards zero.

DIV_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point divide.

result := op1 / op2. Rounds towards zero.

DIV_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer divide.

result := op1 / op2. Rounds towards zero.

DIV_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer divide.

result := op1 / op2. Rounds towards zero.

DIV_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer divide.

result := op1 / op2. Rounds towards zero.

DIV_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer divide.

result := op1 / op2. Rounds towards zero.

DIV_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer divide.

result := op1 / op2. Rounds towards zero.

DIV_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer divide.

result := op1 / op2. Rounds towards zero.

DROP (top ⇒ —)

Drops the top value from the stack.

DROP_N (n ⇒ —)

1 operand byte.
Drops n values from the stack at offset fetched from stream.E.g. for offset=1 and n=2, TOS would be untouched, but NOS and 3OS would be discarded

EQ_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point equals.

comp := op1 = op2.

EQ_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point equals.

comp := op1 = op2.

EQ_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer equals.

comp := op1 = op2.

EQ_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer equals.

comp := op1 = op2.

EQ_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer equals.

comp := op1 = op2.

EQ_P (op1, op2 ⇒ comp)

Comparison operation: pointer equals.

comp := op1 = op2.

EQ_R (op1, op2 ⇒ comp)

Comparison operation: reference equals.

comp := op1 = op2.

EQ_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer equals.

comp := op1 = op2.

EQ_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer equals.

comp := op1 = op2.

EQ_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer equals.

comp := op1 = op2.

EXT_I1 (value ⇒ extended)

Sign extends TOS from to a I1 to a pointer-sized integer.

EXT_I2 (value ⇒ extended)

Sign extends TOS from to a I2 to a pointer-sized integer.

EXT_I4 (value ⇒ extended)

Sign extends TOS from to a I4 to a pointer-sized integer.

EXT_I4 (value ⇒ extended)

Sign extends TOS from to a I4 to a pointer-sized integer.

EXT_U1 (value ⇒ extended)

Zero extends TOS from to a U1 to a pointer-sized integer.

EXT_U2 (value ⇒ extended)

Zero extends TOS from to a U2 to a pointer-sized integer.

EXT_U4 (value ⇒ extended)

Zero extends TOS from to a U4 to a pointer-sized integer.

EXT_U4 (value ⇒ extended)

Zero extends TOS from to a U4 to a pointer-sized integer.

F2D (val ⇒ result)

Converts 32 bit floating point to 64 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

F2I (val ⇒ result)

Converts 32 bit floating point to 32 bit signed integer. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

F2L (val ⇒ result)

Converts 32 bit floating point to 64 bit signed integer. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

FAR_JUMP (ip ⇒ —)

Continue interpretation, with the current abstract machine state, at the IP popped from the stack. FAR_JUMP is intended for unusual flow control in code processors and the like.Warning: This instruction is not supported in compiled code, in order to use jumps in compiled code use JUMP instead.

FIELD_IS_NOT_NULL (object, offset ⇒ value)

Tests whether an object field is null. Equivalent to RLOAD_X 0 EQ_X where X is a R, P or a pointer sized integer.

FIELD_IS_NULL (object, offset ⇒ value)

Tests whether an object field is null. Equivalent to RLOAD_X 0 EQ_X where X is a R, P or a pointer sized integer.

FILE(name) (— ⇒ —)

Declares the source file for this code. Informational only, like #FILE in C.

FULLY_INITIALIZED (object ⇒ —)

Declare TOS object to be fully-initialised.This allows optimisations to be made by the toolkit.Drops TOS as a side effect. TOS must be a reference,it is a (serious) error if TOS objecthas *any* uninitialised reference fields

GC_MALLOC (size ⇒ ref)

Allocates size bytes in the heap leaving reference to allocated space in TOS. GC pass may replace with a faster inline version. Defaults to GC_MALLOC_CALL.

GC_MALLOC_CALL (size ⇒ ref)

Allocates size bytes, via a call to the GC collector. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code.Safe to use, but front-ends should use GC_MALLOC instead.

GC_MALLOC_FAST (size ⇒ ref)

Fast allocates size bytes, ref is 0 if cannot allocate fast. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code.For internal toolkit use only.

GC_SAFE (— ⇒ —)

Declares this point to be a safe point for garbage collection to occur at. GC pass should replace with a custom version. Defaults to GC_SAFE_CALL.

GC_SAFE_CALL (— ⇒ —)

Calls GC to inform it that calling thread is safe for garbage collection. Generally users should use GC_SAFE and allow the toolkit to substitute appropriate inline code.

GE_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point greater than or equals.

comp := op1 $\geq$ op2.

GE_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point greater than or equals.

comp := op1 $\geq$ op2.

GE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than or equals.

comp := op1 $\geq$ op2.

GE_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer greater than or equals.

comp := op1 $\geq$ op2.

GE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than or equals.

comp := op1 $\geq$ op2.

GE_P (op1, op2 ⇒ comp)

Comparison operation: pointer greater than or equals.

comp := op1 $\geq$ op2.

GE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than or equals.

comp := op1 $\geq$ op2.

GE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer greater than or equals.

comp := op1 $\geq$ op2.

GE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than or equals.

comp := op1 $\geq$ op2.

GT_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point greater than.

comp := op1 $>$ op2.

GT_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point greater than.

comp := op1 $>$ op2.

GT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than.

comp := op1 $>$ op2.

GT_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer greater than.

comp := op1 $>$ op2.

GT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than.

comp := op1 $>$ op2.

GT_P (op1, op2 ⇒ comp)

Comparison operation: pointer greater than.

comp := op1 $>$ op2.

GT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than.

comp := op1 $>$ op2.

GT_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer greater than.

comp := op1 $>$ op2.

GT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than.

comp := op1 $>$ op2.

HOP(n) (— ⇒ —)

Jump (unconditionally) to TARGET(n)

I2D (val ⇒ result)

Converts 32 bit signed integer to 64 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

I2F (val ⇒ result)

Converts 32 bit signed integer to 32 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

INSERT (n ⇒ address)

1 operand byte.
Pops count off the stack. Inserts n NULLs into the stack at offset fetched from the instruction stream.Ensures that all inserted values are flushed to memory. Pushes the address of first inserted slot to the stack.

INV_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer bitwise invert.

INV_I8 (op1 ⇒ value)

Unary operation: 64 bit signed integer bitwise invert.

INV_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer bitwise invert.

INV_U4 (op1 ⇒ value)

Unary operation: 32 bit unsigned integer bitwise invert.

INV_U8 (op1 ⇒ value)

Unary operation: 64 bit unsigned integer bitwise invert.

INV_U4 (op1 ⇒ value)

Unary operation: 32 bit unsigned integer bitwise invert.

IP (— ⇒ instruction_pointer)

Pushes the current (interpreter) instruction pointer to TOS.

JUMP (— ⇒ —)

2 operand bytes.
Only valid in bytecode context. Performs VM jump. Jumps by N bytes, where N is the next two-byte value in the instruction stream.

L2D (val ⇒ result)

Converts 64 bit signed integer to 64 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

L2F (val ⇒ result)

Converts 64 bit signed integer to 32 bit floating point. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

L2I (val ⇒ result)

Converts 64 bit signed integer to 32 bit signed integer. This is a convertion, not a cast. It is the value that remains the same, not the bit-pattern.

LADDR(name) (— ⇒ addr)

Pushes the address of the local variable 'name' to TOS.

LE_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point less than or equals.

comp := op1 $\leq$ op2.

LE_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point less than or equals.

comp := op1 $\leq$ op2.

LE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than or equals.

comp := op1 $\leq$ op2.

LE_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer less than or equals.

comp := op1 $\leq$ op2.

LE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than or equals.

comp := op1 $\leq$ op2.

LE_P (op1, op2 ⇒ comp)

Comparison operation: pointer less than or equals.

comp := op1 $\leq$ op2.

LE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than or equals.

comp := op1 $\leq$ op2.

LE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer less than or equals.

comp := op1 $\leq$ op2.

LE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than or equals.

comp := op1 $\leq$ op2.

LINE(n) (— ⇒ —)

Set the source code line number of the source code. Informational only, like #LINE in C.

LOCK (lock $\Rightarrow$ —)

Lock the gvmt-lock pointed to by TOS. Pop TOS.

LOCK_INTERNAL (offset, object $\Rightarrow$ —)

Lock the gvmt-lock in object referred to by TOS at offset NOS. Pop both reference and offset from stack.

LSH_I4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit signed integer left shift.

result := op1 $\ll$ op2.

LSH_I8 (op1, op2 $\Rightarrow$ result)

Binary operation: 64 bit signed integer left shift.

result := op1 $\ll$ op2.

LSH_I4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit signed integer left shift.

result := op1 $\ll$ op2.

LSH_U4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit unsigned integer left shift.

result := op1 $\ll$ op2.

LSH_U8 (op1, op2 $\Rightarrow$ result)

Binary operation: 64 bit unsigned integer left shift.

result := op1 $\ll$ op2.

LSH_U4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit unsigned integer left shift.

result := op1 $\ll$ op2.

LT_F4 (op1, op2 $\Rightarrow$ comp)

Comparison operation: 32 bit floating point less than.

comp := op1 $<$ op2.

LT_F8 (op1, op2 $\Rightarrow$ comp)

Comparison operation: 64 bit floating point less than.

comp := op1 $<$ op2.

LT_I4 (op1, op2 $\Rightarrow$ comp)

Comparison operation: 32 bit signed integer less than.

comp := op1 $<$ op2.

LT_I8 (op1, op2 $\Rightarrow$ comp)

Comparison operation: 64 bit signed integer less than.

comp := op1 $<$ op2.

LT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than.

comp := op1 $<$ op2.

LT_P (op1, op2 ⇒ comp)

Comparison operation: pointer less than.

comp := op1 $<$ op2.

LT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than.

comp := op1 $<$ op2.

LT_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer less than.

comp := op1 $<$ op2.

LT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than.

comp := op1 $<$ op2.

MOD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer modulo.

result := op1

MOD_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer modulo.

result := op1

MOD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer modulo.

result := op1

MOD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer modulo.

result := op1

MOD_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer modulo.

result := op1

MOD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer modulo.

result := op1

MUL_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point multiply.

result := op1 $\times$ op2.

MUL_F8 (op1, op2 $\Rightarrow$ result)

Binary operation: 64 bit floating point multiply.

result := op1 $\times$ op2.

MUL_I4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit signed integer multiply.

result := op1 $\times$ op2.

MUL_I8 (op1, op2 $\Rightarrow$ result)

Binary operation: 64 bit signed integer multiply.

result := op1 $\times$ op2.

MUL_I4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit signed integer multiply.

result := op1 $\times$ op2.

MUL_U4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit unsigned integer multiply.

result := op1 $\times$ op2.

MUL_U8 (op1, op2 $\Rightarrow$ result)

Binary operation: 64 bit unsigned integer multiply.

result := op1 $\times$ op2.

MUL_U4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit unsigned integer multiply.

result := op1 $\times$ op2.

NAME(n,name) (— $\Rightarrow$ —)

Name the nth temporary variable, for debugging purposes.

NARG_F4 (val $\Rightarrow$ —)

Native argument of type 32 bit floating point. TOS is pushed to the native argument stack.

NARG_F8 (val $\Rightarrow$ —)

Native argument of type 64 bit floating point. TOS is pushed to the native argument stack.

NARG_I4 (val $\Rightarrow$ —)

Native argument of type 32 bit signed integer. TOS is pushed to the native argument stack.

NARG_I8 (val $\Rightarrow$ —)

Native argument of type 64 bit signed integer. TOS is pushed to the native argument stack.

NARG_I4 (val ⇒ —)

Native argument of type 32 bit signed integer. TOS is pushed to the native argument stack.

NARG_P (val ⇒ —)

Native argument of type pointer. TOS is pushed to the native argument stack.

NARG_U4 (val ⇒ —)

Native argument of type 32 bit unsigned integer. TOS is pushed to the native argument stack.

NARG_U8 (val ⇒ —)

Native argument of type 64 bit unsigned integer. TOS is pushed to the native argument stack.

NARG_U4 (val ⇒ —)

Native argument of type 32 bit unsigned integer. TOS is pushed to the native argument stack.

NEG_F4 (op1 ⇒ value)

Unary operation: 32 bit floating point negate.

NEG_F8 (op1 ⇒ value)

Unary operation: 64 bit floating point negate.

NEG_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer negate.

NEG_I8 (op1 ⇒ value)

Unary operation: 64 bit signed integer negate.

NEG_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer negate.

NEXT_IP (— ⇒ instruction_pointer)

Pushes the (interpreter) instruction pointer for the *next* instruction to TOS. This is equal to IP plus the length of the current bytecode

NE_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point not equals.

comp := op1 *eq* op2.

NE_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point not equals.

comp := op1 *eq* op2.

NE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer not equals.

comp := op1 *eq* op2.

NE_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer not equals.

comp := op1 *eq* op2.

NE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer not equals.

comp := op1 *eq* op2.

NE_P (op1, op2 ⇒ comp)

Comparison operation: pointer not equals.

comp := op1 *eq* op2.

NE_R (op1, op2 ⇒ comp)

Comparison operation: reference not equals.

comp := op1 *eq* op2.

NE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer not equals.

comp := op1 *eq* op2.

NE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer not equals.

comp := op1 *eq* op2.

NE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer not equals.

comp := op1 *eq* op2.

N_CALL_F4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit floating point.

N_CALL_F8(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 64 bit floating point.

N_CALL_I4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit signed integer.

N_CALL_I4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit signed integer.

N_CALL_I8(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 64 bit signed integer.

N_CALL_NO_GC_F4(n) (— ⇒ value)

As N_CALL_F4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_F8(n) (— ⇒ value)

As N_CALL_F8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I4(n) (— ⇒ value)

As N_CALL_I4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I4(n) (— ⇒ value)

As N_CALL_I4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I8(n) (— ⇒ value)

As N_CALL_I8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_P(n) (— ⇒ value)

As N_CALL_P(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_R(n) (— ⇒ value)

As N_CALL_R(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U4(n) (— ⇒ value)

As N_CALL_U4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U4(n) (— ⇒ value)

As N_CALL_U4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U8(n) (— ⇒ value)

As N_CALL_U8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_V(n) (— ⇒ value)

As N_CALL_V(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_P(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a pointer.

N_CALL_R(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a reference.

N_CALL_U4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit unsigned integer.

N_CALL_U4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit unsigned integer.

N_CALL_U8(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 64 bit unsigned integer.

N_CALL_V(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a void.

OPCODE (— ⇒ opcode)

Pushes the current opcode to TOS.

OR_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise or.

result := op1 | op2.

OR_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer bitwise or.

result := op1 | op2.

OR_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise or.

result := op1 | op2.

OR_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise or.

result := op1 | op2.

OR_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer bitwise or.

result := op1 | op2.

OR_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise or.

result := op1 | op2.

PICK_F4 (— ⇒ $\mathbf{n^{th}}$)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_F8 (— ⇒ $\mathbf{n^{th}}$)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I4 (— ⇒ $\mathbf{n^{th}}$)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I8 (— ⇒ $\mathbf{n^{th}}$)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I4 (— ⇒ $\mathbf{n^{th}}$)

1 operand byte.
Picks the nth item from the data

stack(TOS is index 0)and pushes it to TOS.

PICK_P (— ⇒ n^{th})

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_R (— ⇒ n^{th})

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U4 (— ⇒ n^{th})

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U8 (— ⇒ n^{th})

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U4 (— ⇒ n^{th})

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PIN (object ⇒ pinned)

Pins the object on TOS. Changes type of TOS from a reference to a pointer.

PINNED_OBJECT (pointer ⇒ object)

Declares that pointer is in fact a reference to a pinned object. Changes type of TOS from a pointer to a reference. It is an error if the pointer is not a reference to a pinned object. Incorrect use of this instruction can be difficult to detect. Use with care.

PLOAD_F4 (addr ⇒ value)

Load from memory. Push 32 bit floating point value loaded from address in TOS (which must be a pointer).

PLOAD_F8 (addr ⇒ value)

Load from memory. Push 64 bit floating point value loaded from address in TOS (which must be a pointer).

PLOAD_I1 (addr ⇒ value)

Load from memory. Push 8 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I2 (addr ⇒ value)

Load from memory. Push 16 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I4 (addr ⇒ value)

Load from memory. Push 32 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I8 (addr ⇒ value)

Load from memory. Push 64 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I4 (addr ⇒ value)

Load from memory. Push 32 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_P (addr ⇒ value)

Load from memory. Push pointer value loaded from address in TOS (which must be a pointer).

PLOAD_R (addr ⇒ value)

Load from memory. Push reference value loaded from address in TOS (which must be a pointer).

PLOAD_U1 (addr ⇒ value)

Load from memory. Push 8 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U2 (addr ⇒ value)

Load from memory. Push 16 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U4 (addr ⇒ value)

Load from memory. Push 32 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U8 (addr ⇒ value)

Load from memory. Push 64 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U4 (addr ⇒ value)

Load from memory. Push 32 bit unsigned integer value loaded from address in TOS (which must be a pointer).

POP_STATE (— ⇒ value)

Pops and discards the state-object on top of the state stack.

PSTORE_F4 (value, array ⇒ —)

Store to memory. Store 32 bit floating point value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_F8 (value, array ⇒ —)

Store to memory. Store 64 bit floating point value in NOS to address in TOS.

(TOS must be a pointer)

PSTORE_I1 (value, array ⇒ —)

Store to memory. Store 8 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I2 (value, array ⇒ —)

Store to memory. Store 16 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I4 (value, array ⇒ —)

Store to memory. Store 32 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I8 (value, array ⇒ —)

Store to memory. Store 64 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I4 (value, array ⇒ —)

Store to memory. Store 32 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_P (value, array ⇒ —)

Store to memory. Store pointer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_R (value, array ⇒ —)

Store to memory. Store reference value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U1 (value, array ⇒ —)

Store to memory. Store 8 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U2 (value, array ⇒ —)

Store to memory. Store 16 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U4 (value, array ⇒ —)

Store to memory. Store 32 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U8 (value, array ⇒ —)

Store to memory. Store 64 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U4 (value, array ⇒ —)

Store to memory. Store 32 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PUSH_CURRENT_STATE (— ⇒ value)

Pushes a new state-object to the state stack and pushes 0 to TOS, when initially executed. When execution resumes after a RAISE or TRANSFER, then the value in the transfer register is pushed to TOS.

RAISE (value ⇒ —)

Pop TOS, which must be a reference, and place in the transfer register. Examine the state object on top of state stack. Pop values from the data-stack to the depth recorded. Resume execution from the PUSH_CURRENT_STATE instruction that stored the state object on the state stack.

RETURN_F4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_F8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_P (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_R (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_V (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RLOAD_F4 (object, offset ⇒ value)

Load from object. Load 32 bit floating point value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_F8 (object, offset ⇒ value)

Load from object. Load 64 bit floating point value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I1 (object, offset ⇒ value)

Load from object. Load 8 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I2 (object, offset ⇒ value)

Load from object. Load 16 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I4 (object, offset ⇒ value)

Load from object. Load 32 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I8 (object, offset ⇒ value)

Load from object. Load 64 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I4 (object, offset ⇒ value)

Load from object. Load 32 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_P (object, offset ⇒ value)

Load from object. Load pointer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_R (object, offset ⇒ value)

Load from object. Load reference value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)Any read-barriers required by the garbage collector are performed.

RLOAD_U1 (object, offset ⇒ value)

Load from object. Load 8 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U2 (object, offset ⇒ value)

Load from object. Load 16 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U4 (object, offset ⇒ value)

Load from object. Load 32 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U8 (object, offset ⇒ value)

Load from object. Load 64 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U4 (object, offset ⇒ value)

Load from object. Load 32 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer arithmetic right shift.

result := op1 $\gg$ op2.

RSH_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer arithmetic right shift.

result := op1 $\gg$ op2.

RSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer arithmetic right shift.

result := op1 $\gg$ op2.

RSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer logical right shift.

result := op1 $\gg$ op2.

RSH_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer logical right shift.

result := op1 $\gg$ op2.

RSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer logical right shift.

result := op1 $\gg$ op2.

RSTORE_F4 (value, object, offset ⇒ —)

Store into object. Store 32 bit floating point value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_F8 (value, object, offset ⇒ —)

Store into object. Store 64 bit floating point value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I1 (value, object, offset ⇒ —)

Store into object. Store 8 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I2 (value, object, offset ⇒ —)

Store into object. Store 16 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I4 (value, object, offset ⇒ —)

Store into object. Store 32 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I8 (value, object, offset ⇒ —)

Store into object. Store 64 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I4 (value, object, offset ⇒ —)

Store into object. Store 32 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_P (value, object, offset ⇒ —)

Store into object. Store pointer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_R (value, object, offset ⇒ —)

Store into object. Store reference value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)Any write-barriers required by the garbage collector are performed.

RSTORE_U1 (value, object, offset ⇒ —)

Store into object. Store 8 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U2 (value, object, offset ⇒ —)

Store into object. Store 16 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U4 (value, object, offset ⇒ —)

Store into object. Store 32 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U8 (value, object, offset ⇒ —)

Store into object. Store 64 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U4 (value, object, offset ⇒ —)

Store into object. Store 32 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

SIGN (val ⇒ extended)

On a 32 bit machine, sign extend TOS from a 32 bit value to a 64 bit value. This is a no-op for 64bit machines.

STACK (— ⇒ sp)

Pushes the data-stack stack-pointer to TOS. The data stack grows downwards, so stack items will be at non-negative offsets from sp. Values subsequently pushed on to the stack are not visible. Attempting to access values at negative offsets is an error. As soon as a net positive number of values are popped from the stack, sp becomes invalid and should *not* be used.

SUB_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point subtract.

result := op1 - op2.

SUB_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point subtract.

result := op1 - op2.

SUB_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer subtract.

result := op1 - op2.

SUB_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer subtract.

result := op1 - op2.

SUB_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer subtract.

result := op1 - op2.

SUB_P (op1, op2 ⇒ result)

Binary operation: pointer subtract.

result := op1 - op2.

SUB_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer subtract.

result := op1 - op2.

SUB_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer subtract.

result := op1 - op2.

SUB_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer subtract.

result := op1 - op2.

SYMBOL (— ⇒ address)

2 operand bytes.
Push address of symbol to TOS

TARGET(n) (— ⇒ —)

Target for Jump and Branch.

TLOAD_F4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit floating point

TLOAD_F8(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 64 bit floating point

TLOAD_I4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit signed integer

TLOAD_I4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit signed integer

TLOAD_I8(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 64 bit signed integer

TLOAD_P(n) (— ⇒ value)

Push the contents of the nth temporary variable as a pointer

TLOAD_R(n) (— ⇒ value)

Push the contents of the nth temporary variable as a reference

TLOAD_U4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit unsigned integer

TLOAD_U4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit unsigned integer

TLOAD_U8(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 64 bit unsigned integer

TRANSFER (— ⇒ —)

Pop TOS, which must be a reference, and place in the transfer register. Resume execution from the PUSH_CURRENT_STATE instruction that stored the state object on the state stack. Unlike RAISE, TRANSFER does not modify the data stack.

TSTORE_F4(n) (value ⇒ —)

Pop a 32 bit floating point from the stack and store in the nth temporary variable.

TSTORE_F8(n) (value ⇒ —)

Pop a 64 bit floating point from the stack and store in the nth temporary variable.

TSTORE_I4(n) (value ⇒ —)

Pop a 32 bit signed integer from the stack and store in the nth temporary variable.

TSTORE_I4(n) (value ⇒ —)

Pop a 32 bit signed integer from the stack and store in the nth temporary variable.

TSTORE_I8(n) (value ⇒ —)

Pop a 64 bit signed integer from the stack and store in the nth temporary variable.

TSTORE_P(n) (value ⇒ —)

Pop a pointer from the stack and store in the nth temporary variable.

TSTORE_R(n) (value ⇒ —)

Pop a reference from the stack and store in the nth temporary variable.

TSTORE_U4(n) (value ⇒ —)

Pop a 32 bit unsigned integer from the stack and store in the nth temporary variable.

TSTORE_U4(n) (value ⇒ —)

Pop a 32 bit unsigned integer from the stack and store in the nth temporary variable.

TSTORE_U8(n) (value ⇒ —)

Pop a 64 bit unsigned integer from the stack and store in the nth temporary variable.

TYPE_NAME(n,name) (— ⇒ —)

Name the (reference) type of the nth temporary variable, for debugging purposes.

UNLOCK (lock ⇒ —)

Unlock the gvmt-lock pointed to by TOS. Pop TOS.

UNLOCK_INTERNAL (offset, object ⇒ —)

Unlock the fast-lock in object referred to by TOS at offset NOS. Pop both reference and offset from stack.

V_CALL_F4 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit floating point.

V_CALL_F8 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 64 bit floating point.

V_CALL_I4 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit signed integer.

V_CALL_I8 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 64 bit signed integer.

V_CALL_I4 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit signed integer.

V_CALL_P (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a pointer.

V_CALL_R (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are

from the data stack. The function called must return a reference.

V_CALL_U4 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit unsigned integer.

V_CALL_U8 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 64 bit unsigned integer.

V_CALL_U4 (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit unsigned integer.

V_CALL_V (— ⇒ value)

1 operand byte.
Variadic call. The number of parameters, n, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return void.

XOR_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise exclusive or.

result := op1 $\oplus$ op2.

XOR_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer bitwise exclusive or.

result := op1 $\oplus$ op2.

XOR_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise exclusive or.

result := op1 $\oplus$ op2.

XOR_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise exclusive or.

result := op1 $\oplus$ op2.

XOR_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer bitwise exclusive or.

result := op1 $\oplus$ op2.

XOR_U4 (op1, op2 $\Rightarrow$ result)

Binary operation: 32 bit unsigned integer bitwise exclusive or.

result := op1 $\oplus$ op2.

ZERO (val $\Rightarrow$ extended)

On a 32 bit machine, zero extend TOS from a 32 bit value to a 64 bit value. This is a no-op for 64bit machines.

Appendix B

The GVMT Abstract Machine Language Grammar

Top Level Rule

```
file: section+ debug_info?
```

Rules

```
section: (bytecode_section | code_section | heap_section |
          opaque_section | root_section)

bytecode_section: '.bytecodes' new_lines
                  ((bytecode_directive | bytecode) new_lines)*

code_section: '.code' new_lines
              ((code_directive | function) new_lines)*

heap_section: '.heap' new_lines
              ((heap_directive | data_declaration) new_lines)*

opaque_section: '.opaque' new_lines
              ((data_directive | data_declaration) new_lines)*

roots_section: '.roots' new_lines
               ((data_directive |  address) new_lines)*

instruction: ID ( '(' ( ID ( ',' ID)* )? ')' )?
```

```
bytecode: ID ( '=' digit+ )? ( '[' qualifier* ']' )?  ':' instruction* ';'

function: ID ( '[' qualifier* ']' )? ':' instruction* ';'

data_declaration: integral_value | float_value |
                  string_value | address

bytecode_directive: '.local' | '.name' ID | '.master'

heap_directive: '.public' ID | '.object' ID | '.end'

data_directive: '.public' ID | '.label' ID

integral_value: int_type number

float_value: float_type float_number

string_value: 'string' text

address: 'address' (0 | ID)

debug_info: ((type_directive | member_directive) new_lines)*

type_directive: '.type' ('struct'|'object') ID

member_directive: '.member' ID member_type '@' number

member_type: int_type | float_type | pointer_type | reference_type

struct_type: 'S(' ID ')'

pointer_type: 'P(' (member_type | '?' | struct_type) ')'

reference_type: 'R(' ID ')'
```

Tokens

```
ID: letter(letter|digit)*

number: digit+

float_number: digit ('.' digit)? ('e' ('+'|'-') digit+)?

text: '"' char* '"'
```

```
int_type: 'u'?'int'('8'|'16'|'32'|'64')

float_type: 'float'('32'|'64')

new_lines: '\n'((' '|'\t')*'\n')*
```

Part Tokens

```
letter: [A-Za-z_]

digit: [0-9]

legal_ascii = Any ascii char from code 32 to 126,
              except '\n', '\t', '\', ''', and '"'.

char: legal_ascii | ''\n' | '\t' | '\\' | '\'' | '\"' |
      '\'[0-3][0-7][0-7]
```

Ignored Tokens

```
whitespace: ' '|'\t'

comment: '/' '/' .* '\n'
```

Appendix C

Python Attribute Lookup Semantics

C.1 Definitions

Attribute lookup in Python refers to the syntactic element `obj.attr` where `obj` is any object, and `attr` is a legal name.

Method calls of the `obj.attr()` are treated the same as any other attribute lookup followed by a call. In other words `obj.attr()` $\equiv$ `f()` where `f = obj.attr`.

For the purposes of the algorithms in this appendix, the following are assumed[1]:

- All machine-level object representations have the field *dict*, which may be null.
- All machine-level type representations have the fields $getattribute$, *get*, *set*, *mro* and, since all classes are also objects, *dict*.
- $getattribute$ and *mro* are never null.
- *get* and *set* may be null.
- If *set* is non-null, *get* must be non-null.

The expression $obj \rightarrow attr$ is taken to mean direct access to the field named $attr$ in the underlying representation of the object referred to by obj.The arrow in the expression $obj \rightarrow attr$ is used as it reflects the C syntax for accessing a field of a structure through a pointer.

The fields $getattribute$, *get* and *set* point, if they are non-null, to machine-level functions (not Python functions). The *mro* field points to vector of types defining

[1] VMs are not required to implement things this way; it just makes the algorithms clearer.

attribute lookup order for that type. The first item in the *mro* vector is the type itself, and the last is always `object`, the base type of everything.

The following expressions are used, in descending order of precedence.

- $\mathcal{T}obj$ is a reference to the type object for the type of obj.
- $f(x,y)$ means call the machine-level function f with x and y as its arguments.
- v_i means the ith item in the vector v.
- $d\langle name \rangle$ means to lookup *name* in the dictionary referred to by d.
- $a \equiv b$ means bitwise equivalence (a and b can be pointers or machine integers.
- $a := b$ means copy the value (which will be a pointer) of b into a.

C.2 Lookup Algorithm

Initially the machine-level pointer obj refers to the Python object `obj`. The Python expression `obj.attr` is evaluated as $\mathcal{T}obj \rightarrow getattribute(obj, attr)$. Although this can be overridden by any type, in general it is not, the main exception being class objects.

The default object lookup is shown in Algorithm C.1. A reference to the resulting object will be stored in $result$.

For class objects, that is objects where $\mathcal{T}obj \subseteq type$, attribute lookup is shown in Algorithm C.2.

The `descriptor_lookup` function is defined in Algorithm C.3.

Algorithm C.1 Python Attribute Lookup (Objects)

$cls := \mathcal{T} obj$
$desc := descriptor_lookup(cls, attr)$
if $desc \neq 0$ and $\mathcal{T} desc \rightarrow set \neq 0$ **then**
  $result := \mathcal{T} desc \rightarrow get(obj, cls)$
else
  $d := obj \rightarrow dict$
  if $d \neq 0$ and $d\langle attr \rangle \neq 0$ **then**
    $result := d\langle attr \rangle$
  else if $desc \neq 0$ and $\mathcal{T} desc \rightarrow get \neq 0$ **then**
    $result := \mathcal{T} desc \rightarrow get(obj, cls)$
  else if $desc \neq 0$ **then**
    $result := desc$
  else
    $result := ERROR$
  end if
end if

Algorithm C.2 Python Attribute Lookup (Types)

$desc := descriptor_lookup(cls, attr)$
if $desc \neq 0$ and $\mathcal{T} desc \rightarrow get \neq 0$ **then**
  $result := \mathcal{T} desc \rightarrow get(None, obj)$
else if $desc \neq 0$ **then**
  $result := desc$
else
  $result := ERROR$
end if

Algorithm C.3 Descriptor Lookup

$mro := cls \rightarrow mro$
$i := 0$
repeat
  $t = mro_i$
  $result := t \rightarrow dict\langle attr \rangle$
  $i := i + 1$
until $result \neq 0$ or $t \equiv object$

Appendix D

Surrogate Functions

The follow functions use some HotPy-specific annotations. These annotations are required to ensure correct semantics, and avoid circularity. The annotations are:

The `@pure` annotation only applies to C functions and states that the function has no global side-effects. The `@c_function` annotation only informs the VM that this is a function written in C. The `@method(class, name)` annotation stores this function as a method in the `class` dictionary with the key `name`. The `@_no_trace` annotation indicates that this function should not appear in a trace-back in the event of an exception being raised

D.1 The __new__ method for tuple

```
@_pure
@c_function
def tuple_from_list(cls:type, l:list)−>tuple:
    pass

@method(tuple, '__new__')
def new_tuple(cls, seq):
    if type(seq) is list:
        return tuple_from_list(cls, seq)
    elif type(seq) is tuple:
        return seq
    else:
        l = [x for x in seq]
        return tuple_from_list(cls, l)
del new_tuple
```

D.2 The __call__ method for type

```
@_no_trace
def type_call(cls, *args, **kws):
    obj = cls.__new__(cls, *args, **kws)
    if isinstance(obj, cls):
        obj.__init__(*args, **kws)
    return obj
```

D.3 The Binary Operator

This function implements binary operators. For addition `name` and `rname` would be '__add__' and '__radd__' respectively.

```
def binary_operator(name, rname, op1, op2):
    t1 = type(op1)
    t2 = type(op2)
    if issubclass(t2, t1):
        if rname in t2.__dict__:
            result = t2.__dict__[rname](op2, op1)
            if result is not NotImplemented:
                return result
        if name in t1.__dict__:
            result = t1.__dict__[name](op1, op2)
            if result is not NotImplemented:
                return result
    else:
        if name in t1.__dict__:
            result = t1.__dict__[name](op1, op2)
            if result is not NotImplemented:
                return result
        if rname in t2.__dict__:
            result = t2.__dict__[rname](op2, op1)
            if result is not NotImplemented:
                return result
    _binary_operator_error(t1, t2, name)
```

Appendix E

The HotPy Virtual Machine Bytecodes

All instructions are shown in the GVMT interpreter description format of name followed by stack effect and instruction effect. Values on the left of the — divider are inputs, those on the right are outputs. All outputs go the stack. Inputs come from the stack unless marked with a #, in which case they are fetched from the instruction stream. #x is a one byte value, ##x is a two byte value. #↑ is a pointer sized value.

For example:

truth(R_object o — R_bool b)

Explanatory text follows the stack effect.

E.1 Base Instructions

The instructions listed in this section are those required to express unoptimised Python programs. The output of the source-to-bytecode compiler consists entirely of these bytecodes.

E.1.1 Atomic Instructions

These instructions are treated as atomic by the optimisers. They are recorded directly by tracing and either left intact or removed entirely by subsequent optimisations.

as_tuple(R_object obj — R_tuple t)

`obj` must be a list or a tuple. If it is a list then it is converted to a tuple. Used for passing parameters (on the caller side).

byte (int #n — R_int i)

Pushes an integer (in the range -128 to 127 inclusive) to the stack.

constant(unsigned ##index — R_object object)

Push a constant to TOS.

```
object = sys._getframe().f_code.co_consts[index]
```

copy(R_object x — R_object x, R_object x)

Duplicates TOS

copy_dict(R_dict d — R_dict d)

Replace dictionary in TOS with a shallow copy, used for parameter marshalling.

delete_global(unsigned ##name —)

Delete from globals (module dictionary)

delete_local(unsigned ##name —)

Delete from frame locals (as dictionary)

dict_insert(R_dict d, R_str key, R_object value — R_dict d)

```
d[key] = value
```

Inserts key/value pair into dict, leaving the dict on the stack. Used for parameter marshalling.

dictionary(— R_dict d)

Pushes a new, empty dictionary to the stack.

drop(R_object x —)

Pops (and discards) TOS

empty_tuple(— R_tuple t)

Pushes an empty tuple to the stack.

exit_loop(R_BaseException ex —)

If `ex` is not a StopIteration then reraise `ex`. Used at exit from a loop to differentiate between loop termination and other exceptions.

false(— R_bool f)

Pushes False to the stack.

flip3 (R_object x1, R_object x2, R_object x3 — R_object x3, R_object x2, R_object x1)

Flips the top three values on the stack.

is(R_object o1, R_object o2 — R_bool b)

```
b = o1 is o2
```

line(unsigned ##lineno —)

Set the line number and calls tracing function (if any).

```
sys._getframe().f_lineno = lineno
```

list(uint8_t #count — R_list l)

Remove top count elements from the stack, creating a new list.

list_append(R_list l, R_object o —)

Used in list comprehension, where `l` is guaranteed to be a list.

load_deref(unsigned #depth, unsigned #n — R_object value)

Load a non-local from frame in stack.

load_frame(unsigned #n — R_object value)

Loads value from the nth local variable. Raise an exception if local variable has been assigned. Equivalent to:

```
value = sys._getframe()._array[n]
```

except that `_array` is not visible in python code.

load_global(unsigned ##name — R_object value)

Load from globals (module dictionary)

load_local(unsigned ##name — R_object value)

Load from frame locals (as dictionary)

name(int ##index — R_str name)

Pushes a string from the code-object's name table.

none(— R_NoneType n)

Pushes None to the stack.

nop(—)

No operation

over (R_object x, R_object x1 — R_object x, R_object x1, R_object x)

Pushes a copy of the second value on the stack to the stack.

pack(uint8_t #count — R_tuple t)

Pack the top count elements from the stack into a new tuple.

pack_params(uint8_t #count — R_tuple t, R_dict empty)

Conceptually ike pack, but also pushes an empty dict. Used for parameter marshalling in the common case where there are no named parameters.

pick (int #n — R_object o)

Picks the nth (TOS is index 0) value from the stack

pop_handler(—)

Pops exception-handler.

rotate (R_object x1, R_object x2, R_object x3 — R_object x2, R_object x3, R_object x1)

Rotates the top three values on the stack.

rotate4 (R_object x1, R_object x2, R_object x3, R_object x4 — R_object x2, R_object x3, R_object x4, R_object x1)

Rotates the top four values on the stack.

rrot(R_object x1, R_object x2, R_object x3 — R_object x3, R_object x1, R_object x2)

Counter rotates the top three values on the stack.

slice(R_object o1, R_object o2, R_object o3 — R_slice s)

```
s = slice(o1, o2, o3)
```

Makes a new slice.

store_deref(unsigned #depth, unsigned #n, R_object value —)

Store a non-local to frame in stack.

store_frame(R_object value, unsigned #n —)

Stores value in the nth local variable. Equivalent to:

```
sys._getframe()._array[n] = value
```

except that `_array` is not visible in python code.

store_global(unsigned ##name, R_object value —)

Store to globals (module dictionary)

store_local(unsigned ##name, R_object value —)

Store to frame locals (as dictionary)

subtype(R_type t0, R_type t1 — R_bool b)

```
b = t0 \subseteq t1
```

swap (R_object x, R_object x1 — R_object x1, R_object x)

Exchanges the top two values on the stack

true(— R_bool t)

Pushes True to the stack.

tuple_concat(R_tuple t1, R_tuple t2 — R_tuple t3)

```
t3 = t1 + t2
```

t1 and t3 must be tuples, used for parameter marshalling.

two_copy(R_object x, R_object x1 — R_object x, R_object x1, R_object x, R_object x1)

Duplicates the two values on the stack

type_check(R_object object, R_type cls — R_bool b)

Push True if `object` is an instance of `cls`, False otherwise.

unpack (uint8_t #len, R_object object —)

`object` must be a list or tuple and of length `len`. Unpacks onto the stack.

E.1.2 Compound Instructions

These instructions can be defined in terms of other instructions. For example the `binary` bytecode can be defined in Python, as shown in Appendix D. These bytecodes can be replaced by a call to a function that implements the same functionality. However, this only done during tracing.

binary(uint8_t #index, R_object l, R_object r — R_object value)

Applies binary operator. Operators are stored in a global tuple.

```
value = binary_operator_tuple[index](l, r)
```

contains(R_object item, R_object container — R_object result)

```
result = item in container
```

delete_attr(unsigned ##index, R_object obj —)

Fetches name from the code-object's name table.

```
del obj.name
```

delitem(R_object seq, R_object index —)

```
del seq[item]
```

getitem(R_object seq, R_object index — R_object value)

```
value = seq[index]
```

inplace(uint8_t #index, R_object l, R_object r — R_object value)

Applies inplace operator. Operators are stored in a global tuple.

```
value = inplace_operator_tuple[index](l, r)
```

iter(R_object o — R_object it)

```
it = iter(o)
```

load_attr(unsigned ##index, R_object obj — R_object value)

Fetches name from the code-object's name table.

```
value = obj.name
```

next(R_object it — R_object value)

```
value = next(it)
```

not(R_object b1 — R_bool b2)

```
b2 = not bool(b1)
```

sequence_to_list_or_tuple(R_object obj — R_object l_t)

Convert `obj` to a list, unless it is already a list or tuple, in which case nothing is done.

setitem(R_object value, R_object seq, R_object index —)

```
seq[index] = value
```

store_attr(unsigned ##index, R_object value, R_object obj —)

Fetches name from the code-object's name table.

```
obj.name = value
```

truth(R_object o — R_bool b)

```
b = bool(o)
```

unary(uint8_t #index, R_object o — R_object value)

Apply unary operator (-x, +x, x)

yield(R_object value —)

Yields value to caller context by performing the following: Pops current frame from stack. Sets current ip to value stored in (now current) frame.

E.1.3 Instructions Replaced During Tracing

These instructions are replaced during tracing with a single alternative. Jumps are eliminated and conditional branches are replaced with condiitonal exits.

debug(— R_bool d)

Push value of global constant `__debug__` (either `True` or `False`)

end_loop(int ##offset —)

Jump by offset (to start of loop) Possible start of tracing.

end_protect(int ##offset —)

Pops exception-handler and jumps by offset

f_call(R_object callable, R_tuple args, R_dict kws — R_object value)

Calls callable with args and kws

```
value = callable(*args, **kws)
```

for_loop(int ##offset —)

As protect, but marks a loop rather than a try-except block.

jump(int ##offset —)

Jump by offset.

on_false(int ##offset, R_object o —)

Jump by offset if TOS evaluates to False

on_true(int ##offset, R_object o —)

Jump by offset if TOS evaluates to True

protect(int ##offset —)

Push an exception-handler, which will catch Exception and jump to current ip + offset.

return(R_object val — R_object val)

If in a generator, raise StopIteration. Otherwise, as yield

E.1.4 Instructions Not Allowed in a Trace

The following instructions have complex semantics and are expected to occur only in start-up code. If any of thme are encountered during tracing the trace is abandoned and normal interpretation continues.

import(R_object file — R_object object)

Used for the import statement.

```
object = __import__(file)
```

make_class(int ##name, R_object dict, R_tuple bases — R_type cls)

Make a new class

make_closure(uint8_t #code_index, R_tuple defaults, R_dict annotations — R_object f)

Make a new closure, code-object is fetched from constant array.

make_func(uint8_t #code_index, R_tuple defaults, R_dict annotations — R_object f)

Make a new function, code-object is fetched from constant array.

new_scope(—)

Creates a frame and pushes it. Used in class declarations

pop_scope(— R_dict locals)

Pops the frame pushed by `new_scope`, leaving its locals dictionary on the stack.

raise(R_object o —)

Raise an exception; o if it is an exception, an error otherwise.

E.2 Instructions Required for Tracing

The instructions required for tracing are mainly equivalents of branch instructions that exit the trace instead. For example the `on_true` bytecode which branhces if the TOS evaluates as true will be replaced with `exit_on_false` if the branch was taken or `exit_on_true` if it was not.

check_valid(R_exec_link link —)

If trace is invalidated, exit trace to unoptimised code.

exit_on_false(R_bool cond, intptr_t #↑exit —)

Exit if `cond` is False; `cond` must be a boolean.

exit_on_true(R_bool cond, intptr_t #↑exit —)

Exit if `cond` is True; `cond` must be a boolean.

fast_constant(unsigned #↑address — R_object object)

Pushes constant object at address. Used by optimiser.

fast_frame(uint8_t #count, intptr_t #↑func, intptr_t #↑next_ip —)

Create and push a new frame for the function `func` and initialise it with the top `count` values on the stack.

fast_line(unsigned ##lineno —)

Set the line number (does not call tracing function)

```
sys._getframe().f_lineno = lineno
```

func_check(intptr_t #↑code, intptr_t #↑exit, R_object obj —)

Ensure that the obj is exactly the function specified by `func`. If it is a different value then exit the trace.

gen_check(unsigned #↑next_ip, intptr_t #↑original_ip, R_generator gen —)

Ensure that `gen` is a generator and that the next ip for the generator is as expected. If not then resume interpretation of unoptimised code.

gen_enter(unsigned #↑caller_ip, intptr_t #↑original_ip, R_generator gen —)

Set the return address in current frame to `caller_ip`, and push generator frame.

gen_exit (—)

Raise a StopIteration exception.

gen_yield(unsigned #↑next_ip, R_object val — R_object val)

Set the current frame's instruction pointer (for resuming the generator) to `next_ip`. Pops current frame from stack. Sets current ip to value stored in previous frame.

init_frame(R_function func, R_tuple t, R_dict d —)

Initialises the current frame from `func`, `t` and `d`. `func` determines number and format of parameters, as well as default values. `t` and `d` contain the parameter values.

interpret(intptr_t #↑resume_ip —)

Resume the interpreter from resume_ip.

load_special(R_object obj, unsigned #index — R_object attr)

Load special attribute, fetching the name from special_name table, `name = special_names[index]`.

```
attr = obj.name
```

There is a fallback function for each index, which is called in the event of obj.name not being defined.

```
attr = fallback[index](obj)
```

make_frame(intptr_t #↑ret_addr, R_function func —)

Set instruction pointer of current frame to `ret_addr`. Create a new frame, determining size from `func`. Push new frame to frame stack.

new_enter(unsigned #↑func_addr, R_type cls, R_tuple t, R_dict d — R_function func, R_tuple t, R_dict d)

Enter the surrogate 'new' function. Replaces `cls` with the surrogate function `func`, replaces `t` with `(cls,) + t` and leaves `d` untouched. Equivalent to:
`flip3` `pack 1` `swap` `tuple_concat` `load_const` `flip3`

pop_frame(—)

Pops frame.

prepare_bm_call(R_bound_method bm, R_tuple t, R_dict d — R_object func, R_tuple t, R_dict d)

Prepare a call for a bound-method. Extracts self and callable from `bm`; prefixing `t` with self.

```
t = (bm.__self__,) + t; func = bm.__func__
```

protect_with_exit(#↑link —)

Push an exception-handler, which will catch Exception and exit to link.

recursion_exit(intptr_t #↑next_ip, intptr_t #↑exit —)

Set next_ip and exits trace.

return_exit(intptr_t #↑exit —)

Pops frame and exits trace.

trace_exit(intptr_t #↑exit —)

Exits trace.

trace_protect(#↑addr —)

Push an exception-handler, which will catch Exception and interpret from `addr`.

type(R_object object — R_type t)

t = type(object)

E.3 Specialised Instructions

Specialised instructions are used when the type of the operands are known. Many are of the form `i_xxx` or `f_xxx` which are operations specialised for integers and floats respectively. The `native_call` instruction allows C functions to be called directly inplace of the `f_call` or `binary` bytecodes, when the tyes are known.

bind(intptr_t #↑func, R_object self — R_bound_method bm)

Create a bound-method from `self` and `func`.

```
bm.__self__ = self;  bm.__func__ = func
```

check_keys(unsigned ##dict_offset, unsigned #↑key_address, intptr_t #↑exit, R_object obj —)

Ensure that the dict-keys of `obj` matches the expected one. If it does not then leave the trace to the handler pointed to by exit. Requires that the type of `obj` is known.

deoptimise_check(intptr_t #↑trace_addr, intptr_t #↑original_ip —)

If trace has been invalidated, resume interpretation from `original_ip`

ensure_initialised(unsigned #n, intptr_t #↑exit —)

If local variable `n` is uninitialised then resume interpreter from `exit`.

ensure_tagged(intptr_t #↑exit, R_object obj — R_object obj)

Ensure that `obj` is a tagged integer. Leaves `obj` on the stack. If it has another type then leave the trace to the handler pointed to by exit.

ensure_tagged2(intptr_t #↑exit, R_object obj, R_object tos — R_object obj, R_object tos)

Like ensure_tagged, but for the second value on the stack. Important for binary operations.

ensure_tagged_drop(intptr_t #↑exit, R_object obj —)

Like ensure_tagged, but does not leave `obj` on the stack.

ensure_type(unsigned #↑code, intptr_t #↑exit, R_object obj — R_object o)

Ensure that `obj` has the type specified by code. Leaves `obj` on the stack. If it has another type then leave the trace to the handler pointed to by exit.

ensure_type2(intptr_t #↑code, intptr_t #↑exit, R_object obj, R_object tos — R_object o, R_object tos)

Like type_ensure, but for the second item on the stack. Important for binary operations.

ensure_type_drop(intptr_t #↑code, intptr_t #↑exit, R_object obj —)

Like ensure_type, but does not leave `obj` on the stack.

ensure_value(intptr_t #↑code, intptr_t #↑exit, R_object obj — R_object o)

Ensure that the obj is exactly value specified by `code`. Leaves obj on the stack. If it is a different value then exit the trace.

f_add(R_float f1, R_float f2 — R_float result)

Addition specialised for floats. `f1` and `f2` must be floats.

```
result = f1 + f2
```

f_div(R_float f1, R_float f2 — R_float result)

Diivision specialised for floats. `f1` and `f2` must be floats.

f_eq(R_float f1, R_float f2 — R_bool result)

Equality test specialised for floats . `f1` and `f2` must be floats.

f_ge(R_float f1, R_float f2 — R_bool result)

Comparison specialised for floats . `f1` and `f2` must be floats.

f_gt(R_float f1, R_float f2 — R_bool result)

Comparison specialised for floats . `f1` and `f2` must be floats.

f_le(R_float f1, R_float f2 — R_bool result)

Comparison specialised for floats . `f1` and `f2` must be floats.

f_lt(R_float f1, R_float f2 — R_bool result)

Comparison specialised for floats . `f1` and `f2` must be floats.

f_mul(R_float f1, R_float f2 — R_float result)

Multiplication specialised for floats. `f1` and `f2` must be floats.

f_ne(R_float f1, R_float f2 — R_bool result)

Inequality test specialised for floats . `f1` and `f2` must be floats.

f_neg(R_float f — R_float result)

Negation specialised for floats. `f` must be a float.

f_sub(R_float f1, R_float f2 — R_float result)

Subtraction specialised for floats. `f1` and `f2` must be floats.

fast_load_attr(unsigned ##dict_offset , unsigned ##index, R_object object — R_object value)

Rapidly loads a value from object dictionary. Requires that both the type of `obj` is known and that its dict-keys have been checked.

fast_load_frame(uintptr_t #n — R_object value)

Loads value from the nth local variable. Like `load_frame`, but does not check that local variable has been assigned.

fast_load_global(intptr_t #↑address, unsigned ##index — R_object value)

Fetch the `dict_values` object from address. The `dict_values` object will belong to a module-level dictionary. Fetch `value` from `index` in the `dict_values` object. Requires guards on the module dict to ensure that dict is not resized or that item is not deleted.

```
value = ((R_dict_values)address)->values[index];
```

fast_not(R_bool b1 — R_bool b2)

```
b2 = not b1
```

b1 must be a boolean.

fast_store_attr(unsigned ##dict_offset, unsigned ##index, R_object value, R_object object —)

Rapidly stores a value to the object dictionary. Requires that both the type of `obj` is known and that its dict-keys have been checked.

fast_store_global(intptr_t #↑address, unsigned ##index, R_object value —)

Stores a global from module dict-values at address, with offset index. Requires guards on the module dict to ensure that dict is not resized.

i2d(R_object o — double out)

Convert a tagged int to a C double (an unboxed float)

i2f(R_object o — R_float result)

Convert a tagged int to a (boxed) float.

i_add(R_int i1, R_int i2, intptr_t #↑exit — R_int result)

Addition specialised for tagged integers. `i1` and `i2` must be tagged integers. If result overflows then box the result and leave the trace to the handler pointed to by exit.

```
result = i1 + i2
```

i_comp_eq(R_int i1, R_int i2 — R_bool result)

Equality test for tagged integers.

i_comp_ge(R_int i1, R_int i2 — R_bool result)

Comparison for tagged integers.

i_comp_gt(R_int i1, R_int i2 — R_bool result)

Comparison for tagged integers.

i_comp_le(R_int i1, R_int i2 — R_bool result)

Comparison for tagged integers.

i_comp_lt(R_int i1, R_int i2 — R_bool result)

Comparison for tagged integers.

```
result = i1 < i2
```

i_comp_ne(R_int i1, R_int i2 — R_bool result)

Inequality test for tagged integers.

i_dec(R_int i1, unsigned #i2, intptr_t #↑exit — R_int result)

Like i_inc, but for subtraction.

```
result = i1 - i2
```

i_div(R_int i1, R_int i2 — R_float result)

```
result = i1 / i2
```

i_inc(R_int i1, unsigned #i2, intptr_t #↑exit — R_int result)

Increment for tagged integers. `i1` must be a tagged integer. If result overflows then box the result and leave the trace to the handler pointed to by exit.

```
result = i1 + i2
```

i_mul(R_int i1, R_int i2 — R_int result)

Multiplies the tagged integers, `i1` and `i2`. Result may be tagged or boxed.

```
result = i1 * i2
```

i_prod(R_int i1, unsigned #i2, intptr_t #↑exit — R_int result)

Multiplies the integers i1 and i2. i1 must be a tagged integer. Result may be tagged or boxed.

```
result = i1 * i2
```

i_rshift(R_int o1, R_int o2 — R_int result)

Right shift i1 by i2. i1 and i2 must be tagged integers.

```
result = i1 >> i2
```

i_sub(R_int i1, R_int i2, intptr_t #↑exit — R_int result)

Like i_add, but for subtraction.

```
result = i1 - i2
```

load_slot(unsigned #offset, R_object object — R_object value)

Load value from object at offset. Raise exception if slot is uninitialised.

native_call(int #count, intptr_t #↑func_addr — R_object value)

Call the native (GVMT) function at func_addr with count parameters.

native_call_no_prot(int #count, intptr_t #↑func_addr — R_object value)

As native_call. The "no_prot" is to inform the optimisers that this function will not raise an exception and does not need to be protected.

native_call_protect(int #count, intptr_t #↑func_addr, intptr_t #↑on_except — R_object value)

Call the native (GVMT) function at func_addr with count parameters. If an exception is raised, resume interpreter from on_except.

native_setitem(intptr_t #↑func_addr, R_object value, R_object seq, R_object index —)

Like native_call, but takes same inputs as setitem and discards return value.

store_slot(unsigned #offset, R_object value, R_object object —)

Store value into object at offset.

unpack_native_params(intptr_t #↑func_addr, R_object c, R_tuple t, R_dict d —)

Unpacks the parameters in t (d must be empty) onto the stack, providing the number of parameters is the same as that required by the builtin (C) function at func_addr. If parameters do not match, raise an exception.

E.4 D.O.C. Instructions

These instructions are those required by the Deferred Object Creation pass. They are either related to unboxing floating point operations, or to storing values in the (thread-local) cache, in order to avoid creating frames.

check_initialised(unsigned #n —)

If local variable `n` is uninitialised then raise an exception.

clear_cache(uintptr_t #count —)

Clears (sets to NULL to allow the objects to be collected) the first `count` cache slots.

d2f(double x — R_float result)

Box a C double to produce a float.

d_add(double l, double r — double out)

```
out = l + r
```

Specialised form for unboxed floats (C doubles).

d_byte(int #val — double out)

Pushes `val` (small integer) as a double.

d_div(double l, double r — double out)

Specialised form for unboxed floats (C doubles).

d_idiv(R_int o1, R_int o2 — double out)

```
out = o1/o2
```

Produce a double by dividing tagged integers.

d_mul(double l, double r — double out)

Specialised form for unboxed floats (C doubles).

d_neg(double f — double out)

Specialised form for unboxed float (C double).

d_sub(double l, double r — double out)

Specialised form for unboxed floats (C doubles).

f2d(R_float f — double out)

Unbox a float to produce a double.

load_from_cache(uintptr_t #n — R_object value)

Loads the nth cached slot. The cache is used to store values that would be stored in the frame, but cannot as the frame is deferred.

store_to_cache(uintptr_t #n, R_object value —)

Stores `value` to n^{th} cached slot.

E.5 Super Instructions

Super-instructions are concatenations of other instructions. For example, the instruction `line_none` is the concatenation of the instructions `line` and `none`.

drop_under (R_object nos, R_object tos — R_object tos)

Drops nos leaving TOS in place.

i_exit_eq(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` = `i2`, for tagged integers.

i_exit_ge(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` ≥ `i2`, for tagged integers.

i_exit_gt(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` > `i2`, for tagged integers.

i_exit_le(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` ≤ `i2`, for tagged integers.

i_exit_lt(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` < `i2`, for tagged integers.

i_exit_ne(R_int i1, R_int i2, intptr_t #↑exit —)

Exit trace if `i1` *eq* `i2`, for tagged integers.

line_byte(—)

Super instruction equal to `line` followed by `byte`

line_fast_constant(—)

Super instruction equal to `line` followed by `fast_constant`

line_fast_load_frame(—)

Super instruction equal to `line` followed by `fast_load_frame`

line_fast_load_global(—)

Super instruction equal to `line` followed by `fast_load_global`

line_load_frame(—)

Super instruction equal to `line` followed by `load_frame`

line_load_global(—)

Super instruction equal to `line` followed by `load_global`

line_none(—)

Super instruction equal to `line` followed by `none`

Appendix F

Results

	gcbench	pystone	richards	fannkuch	fasta	spectral
HotPy (base, C)	1.06	0.78	0.52	0.40	0.83	0.90
HotPy (base, Py)	1.08	1.02	0.53	1.25	1.21	0.90
HotPy (JIT, C)	0.55	0.42	0.37	0.31	0.43	0.31
HotPy (JIT, Py)	0.55	0.43	0.38	0.29	0.46	0.31
HotPy (int-opt, C)	0.41	0.33	0.25	0.25	0.53	0.35
HotPy (int-opt, Py)	0.41	0.33	0.27	0.28	0.55	0.35
HotPy(C) t	1.05	0.71	0.51	0.34	0.74	0.75
HotPy(C) tc	1.27	1.34	0.80	0.52	0.98	1.01
HotPy(C) td	1.20	0.82	0.54	0.43	0.87	0.83
HotPy(C) tdc	1.31	1.23	0.82	0.64	0.93	0.98
HotPy(C) ts	0.71	0.41	0.25	0.21	0.48	0.38
HotPy(C) tsc	0.97	0.59	0.38	0.27	0.51	0.47
HotPy(C) tsd	0.41	0.33	0.25	0.25	0.53	0.35
HotPy(C) tsdc	0.54	0.42	0.37	0.32	0.43	0.31
HotPy(Py) t	1.07	0.93	0.53	1.20	1.08	0.76
HotPy(Py) tc	1.28	1.99	0.83	1.79	1.35	1.03
HotPy(Py) td	1.22	1.03	0.56	1.26	1.18	0.84
HotPy(Py) tdc	1.34	1.89	0.84	1.77	1.24	0.95
HotPy(Py) ts	0.73	0.60	0.28	0.81	0.74	0.38
HotPy(Py) tsc	1.03	0.81	0.41	0.82	0.67	0.47
HotPy(Py) tsd	0.41	0.33	0.27	0.33	0.55	0.35
HotPy(Py) tsdc	0.55	0.43	0.38	0.29	0.47	0.31
Python3	1.61	1.02	0.60	0.42	0.43	0.75
PyPy (interpreter)	2.34	1.66	1.63	0.46	0.92	0.86
PyPy (with JIT)	1.10	0.36	0.68	0.19	0.43	0.23
Un. Sw. (always)	2.68	2.64	3.24	0.75	1.00	0.83
Un. Sw. (default)	1.51	2.13	1.60	0.62	0.51	0.71
Un. Sw. (no JIT)	1.61	0.86	0.89	0.32	0.32	0.58

Table F.1: Timings (in seconds); short benchmarks.

	gcbench	pystone	richards	fannkuch	fasta	spectral
HotPy (base, C)	9.79	7.32	4.69	3.49	7.79	7.59
HotPy (base, Py)	9.96	9.65	4.90	12.46	11.69	7.59
HotPy (JIT, C)	2.80	1.25	2.21	1.44	1.84	1.43
HotPy (JIT, Py)	2.68	1.27	2.42	1.14	2.04	1.44
HotPy (int-opt, C)	3.51	3.07	2.17	2.28	5.09	2.84
HotPy (int-opt, Py)	3.50	3.09	2.37	2.27	5.23	2.84
HotPy(C) t	9.63	6.88	4.89	3.22	7.17	6.58
HotPy(C) tc	9.70	7.85	6.30	3.56	6.22	6.03
HotPy(C) td	11.01	8.00	5.18	4.04	8.53	7.26
HotPy(C) tdc	10.12	8.17	6.40	3.76	6.01	5.47
HotPy(C) ts	6.53	3.82	2.23	1.87	4.55	3.11
HotPy(C) tsc	5.64	2.43	2.39	1.37	2.72	2.79
HotPy(C) tsd	3.50	3.07	2.17	2.26	5.08	2.84
HotPy(C) tsdc	2.79	1.24	2.21	1.44	1.83	1.43
HotPy(Py) t	9.85	9.04	5.07	11.79	10.56	6.59
HotPy(Py) tc	9.75	11.22	6.67	12.71	8.77	6.13
HotPy(Py) td	11.37	10.03	5.45	12.59	11.59	7.32
HotPy(Py) tdc	10.19	11.01	6.72	11.36	8.04	5.58
HotPy(Py) ts	6.61	5.81	2.48	7.77	7.17	3.13
HotPy(Py) tsc	5.92	3.98	2.93	4.59	3.95	2.79
HotPy(Py) tsd	3.52	3.08	2.35	2.31	5.26	2.84
HotPy(Py) tsdc	2.69	1.27	2.44	1.15	2.03	1.43
Python3	15.07	9.82	5.62	3.88	3.92	6.66
PyPy (interpreter)	22.74	16.33	16.08	4.44	8.98	7.33
PyPy (with JIT)	3.95	1.36	1.43	0.93	3.61	0.57
Un. Sw. (always)	14.36	10.62	11.44	2.49	3.13	3.99
Un. Sw. (default)	12.48	14.28	12.87	5.91	2.96	3.18
Un. Sw. (no JIT)	15.15	8.33	8.56	3.04	2.89	4.93

Table F.2: Timings (in seconds); medium benchmarks.

	gcbench	pystone	richards	fannkuch	fasta	spectral
HotPy (JIT, C)	27.17	7.27	13.15	11.72	13.45	10.18
HotPy (JIT, Py)	24.52	7.66	13.46	8.38	15.08	10.20
HotPy (int-opt, C)	38.65	30.41	21.32	24.85	44.11	30.78
HotPy (int-opt, Py)	38.93	30.57	23.87	24.53	45.95	30.86
HotPy(C) t	108.11	68.63	49.49	34.79	72.73	71.59
HotPy(C) tc	101.87	72.72	52.41	31.50	59.35	61.45
HotPy(C) td	119.76	80.09	52.94	44.72	84.75	80.64
HotPy(C) tdc	106.67	75.30	52.32	34.31	55.02	54.87
HotPy(C) ts	74.97	38.01	22.02	20.19	39.50	33.85
HotPy(C) tsc	57.94	18.25	15.49	12.11	21.59	24.00
HotPy(C) tsd	38.75	30.41	21.29	24.89	44.09	30.79
HotPy(C) tsdc	27.30	7.29	13.10	11.74	13.49	10.12
HotPy(Py) t	110.01	90.21	51.40	132.25	105.88	72.98
HotPy(Py) tc	104.70	99.77	54.94	113.59	83.04	62.89
HotPy(Py) td	122.37	99.80	55.14	140.84	115.13	79.64
HotPy(Py) tdc	107.51	98.63	54.87	100.63	76.02	56.54
HotPy(Py) ts	77.93	57.79	24.64	87.01	63.02	34.01
HotPy(Py) tsc	61.80	31.81	16.71	41.86	31.79	24.04
HotPy(Py) tsd	38.92	30.58	23.60	25.02	46.11	30.83
HotPy(Py) tsdc	24.61	7.62	13.46	8.21	15.08	10.14
Python3	239.55	98.44	55.69	42.63	39.76	73.89
PyPy (with JIT)	32.75	10.93	8.16	9.29	34.88	5.92
Un. Sw. (always)	220.21	90.34	92.79	22.56	24.68	40.45
Un. Sw. (default)	211.84	135.72	122.90	64.51	25.21	42.57

Table F.3: Timings (in seconds); long benchmarks.

Bibliography

[1] Gnu lightning. `http://www.gnu.org/software/lightning/`.

[2] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA*, pages 207–222, 1999.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge Univ. Press, 1991.

[5] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. Research Report RC23143 (W0312-097), IBM, 2004.

[6] John Aycock. A brief history of just-in-time. *CSURV: Computing Surveys*, 35, 2003.

[7] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for java. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI*, pages 1–12, 2000.

[9] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[10] E. C. Berkeley and Daniel G. Bobrow. The programming language LISP: Its operation and applications. Report, The MIT Press, Cambridge, Massachusetts, 1964.

[11] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Code Generation and Optimization (CGO)*, pages 15–26, 2005.

[12] R. E. Berry. Experience with the pascal P-compiler. *Software – Practice and Experience*, 8(5):617–627, September 1978.

[13] Bigloo homepage. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`.

[14] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[15] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Edinburgh, May 2004.

[16] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In David F. Bacon and Amer Diwan, editors, *Proceedings of the Fourth ISMM*, pages 143–151, Vancouver, Canada, October 2004. ACM Press.

[17] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[18] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

[19] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing JIT compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM.

[20] Carl Friedrich Bolz and Armin Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[21] Kevin Casey, David Gregg, and M. Anton Ertl. Tiger - an interpreter generation tool. In Rastislav Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2005.

[22] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, March 1992.

[23] Computer Language Shootout. `http://shootout.alioth.debian.org/u32/ruby.php`.

[24] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[25] Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Comp. Syst*, 16(7):739–751, 2000.

[26] Mark Dufour. Shed skin — an optimizing Python-to-C++ compiler. Master's thesis, Delft University of Technology, 2006.

[27] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University Computer Science Department, March 1994.

[28] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen - a generator of efficient virtual machine interpreters. *Softw, Pract. Exper*, 32(3):265–294, 2002.

[29] Maciej Fijalkowski. `http://pycon.blip.tv/file/3259650/`.

[30] Christopher W. Fraser and David R. Hanson. The lcc 4.x code-generation interface. Technical Report MSR-TR-2001-64, Microsoft Research (MSR), July 2001.

[31] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[32] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.

[33] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. A lazy developer approach: building a JVM with third party software. In Luís Veiga, Vasco Amaral, R. Nigel Horspool, and Giacomo Cabri, editors, *PPPJ*, volume 347 of *ACM International Conference Proceeding Series*, pages 73–82. ACM, 2008.

[34] GNU classpath. `http://www.gnu.org/software/classpath/`.

[35] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995.

[36] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 150–156, New York, NY, USA, 2002. ACM.

[37] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD dissertation, Stanford University, Stanford , CA , USA, 1994.

[38] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[39] Richard L. Hudson, J. E Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical report, Amherst, MA, USA, 1991.

[40] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The implementation of Lua 5.0. *J. UCS*, 11(7):1159–1176, 2005.

[41] IronPython homepage. `http://ironpython.codeplex.com/`.

[42] Java. `http://java.sun.com/`.

[43] The Jikes research virtual machine. `http://jikesrvm.sourceforge.net/`.

[44] Richard Jones. The garbage collection bibliography. `http://www.cs.kent.ac.uk/people/staff/rej/gcbib/gcbib.html`.

[45] Richard Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[46] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 92.

[47] Guy Lewis Steele jr. Data representations in PDP-10 MACLISP. Report A. I. MEMO 420, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1977.

[48] Dong-Heon Jung, Sung-Hwan Bae, Jaemok Lee, Soo-Mook Moon, and JongKuk Park. Supporting precise garbage collection in java bytecode-to-c ahead-of-time compiler for embedded systems. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 35–42, New York, NY, USA, 2006. ACM.

[49] The Jython Project. `http://jython.org`.

[50] Chris Arthur Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[51] The LuaJIT project. `http://luajit.org/`.

[52] Martin Maierhofer and M. Anton Ertl. Local stack allocation. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 189–203, London, UK, 1998. Springer-Verlag.

[53] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 11–20, New York, NY, USA, 2008. ACM.

[54] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. Assoc. Comput. Mach.*, 3(3):184–195, 1960.

[55] Erik Meijer and John Gough. Technical overview of the common language runtime. Technical report, Microsoft Research, 2000.

[56] C. H. Moore. FORTH: a new way to program a mini computer. *Astronomy & Astrophysics Supplement Series*, 15:497–511, April–June 1974.

[57] PLT scheme. `http://plt-scheme.org/`.

[58] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Java™ Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[59] Mike Pall. http://www.nntp.perl.org/group/perl.perl6.internals/2007/09/msg40359.html.

[60] Parrot Virtual Machine. `http://www.parrot.org/`.

[61] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *ISMM*, pages 143–154, 2000.

[62] Python Software Foundation. Python programming language. `http://www.python.org/`.

[63] B. Randell and L. J. Russell. *Algol 60 implementation*. Academic Press, New York, NY, 1964.

[64] Martin Richards. BCPL: A tool for compiler writing and system programming. In *Proceedings AFIPS Spring Joint Computer Conference, Boston, Mass.*, pages 557–566. American Federation of Information Processing Societies, May 1969.

[65] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for Python. In Nevin Heintze and Peter Sestoft, editors, *PEPM*, pages 15–26. ACM, 2004.

[66] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 944–953. ACM, 2006.

[67] Ruby programming language. `http://www.ruby-lang.org/`.

[68] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 285–293, New York, NY, USA, 1988. ACM.

[69] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. In *Virtual Execution Environments (VEE '05)*, pages 153–163, 2005.

[70] James E. Smith and Ravi Nair. *Virtual Machines*. Morgan Kaufmann, 2005.

[71] Patrick Sobalvarro. A lifetime-based garbage collector for lisp systems on general-purpose computers. Technical Report AITR-1417, MIT, AI Lab, February 1988.

[72] Jr. Steele, Guy Lewis and Gerald Jay Sussman. The revised report on scheme: A dialect of lisp. Technical Report AI Memo 452, Massachusetts Institute of Technology, 1978.

[73] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing Highly Flexible Virtual Machines: the JnJVM Experience. *Software: Practice and Experience (SPE)*, 2008.

[74] David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[75] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 11–20. ACM, 2005.

[76] Andrew Whitworth. http://wknight8111.blogspot.com/2009/10/optimizing-parrot.html.

[77] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *CGO*, pages 278–287. ACM, 2010.

[78] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. YETI: a graduallY Extensible Trace Interpreter. In Chandra Krintz, Steven Hand, and David Tarditi, editors, *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007*, pages 83–93. ACM, 2007.