Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DISSERTATION

Purely Interpretative Optimizations

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Univ. Prof. Dipl.-Inf. Dr. rer. nat. Jens Knoop
E185/1
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. (FH) Stefan Brunthaler
Matr.-Nr.: 0457299
brunthaler@complang.tuwien.ac.at
Giessaufgasse 4/11
1050 Wien

Diese Dissertation haben begutachtet:

Prof. Dr. Jens Knoop	Prof. Dr. Michael Franz

Wien, am 14. Februar 2011

Stefan Brunthaler

Kurzfassung

Interpretierer sind einfach zu implementieren und können mit marginalem Zusatzaufwand portabel gemacht werden. Daher werden viele populäre Programmiersprachen interpretiert und nicht traditionell kompiliert. Im Allgemeinen sind diese Eigenschaften von Interpretierern vorteilhaft, jedoch verfügen Interpretierer über einen gravierenden Nachteil: sub-optimale Ausführungsgeschwindigkeit. Glücklicherweise wurde 1984 von L. Peter Deutsch und Allan Schiffman das Verfahren zur dynamischen Übersetzung eingeführt und in weiterer Folge popularisiert, sodass wir heute über hoch-performante dynamische Übersetzter, wie zum Beispiel die Java virtual machine oder die .NET Umgebung verfügen. Unglücklicherweise bedeutet die Entscheidung einen dynamischen Übersetzer zu implementieren gleichzeitig auch, die ursprünglichen Eigenschaften eines Interpretierers, mithin die Einfachheit der Implementierung und deren Portabilität, größtenteils aufzugeben.

Die vorliegende Dissertation a) klärt auf warum für bestimmte Arten von Interpretierern bereits bekannte Techniken nicht ihr übliches Potential entfalten, b) gibt eine Orientierung welche Arten von Optimierungen höheres Potential bieten und c) präsentiert mehrere rein-interpretative Optimierungstechniken mit substantiellem Optimierungs-Potential – Geschwindigkeitssteigerungen bis zu einem Faktor von 2.4176 sind möglich – jedoch ohne die Eigenschaften von Interpretierern zu beeinträchtigen.

Abstract

Interpreters are easy to implement and can be made portable with only little extra effort. Therefore, many popular programming languages choose an interpreter instead of a compiler as an execution platform. While the characteristics of interpreters are regarded as an upside, usually, their major downside is considered to be sub-optimal performance. Fortunately, in 1984 L. Peter Deutsch and Allan Schiffman introduced the modern concept of dynamic compilation sub-systems to the programming language implementation community, which subsequently became a success story, resulting in today's high performance just-in-time compilers for Java and .NET. Unfortunately, on the other hand, deciding to implement a dynamic compilation sub-system involves trading off the valuable innate characteristics of interpreters.

This dissertation a) explains why for some interpreters known techniques do not yield reported speedups, b) provides orientation for focusing on other optimization targets, and c) presents several purely-interpretative optimization techniques that result in substantial speedups—we report speedups of up to 2.4176—while simultaneously preserving the ease of implementation and portability characteristics.

[Disclaimer: All trademarks are the property of their respective owners.]

Acknowledgments

Parts of this thesis were previously published at the following conferences (in chronological order): BYTECODE'09 [Bru09], SAC'10 [Bru10a], ECOOP'10 [Bru10c], DLS'10 [Bru10b], and the upcoming CC'11 [Bru11]. The material presented herein has been updated and extended. Where necessary, material has been rewritten to improve the presentation in general—and the reading experience in particular. I deeply appreciate all the constructive reviews I got from the anonymous reviewers from all of these conferences. As far as I remember, almost all of them were encouraging and provided valuable insights—one way or another.

Prof. Jens Knoop deserves most of the credits from an advisory perspective: I feel I have learned most of the tools and skills necessary for succeeding in such a highly competitive endeavor as research nowadays is. He always took the time to diligently review even the earliest drafts of my papers and actively supported every idea I had along the way: getting internships—though ultimately not going—, getting funding for travel expenses and writing an ultimately successful grant proposal for the Austrian Science Fund (FWF). I have been very fortunate to have him as my adviser.

Besides Prof. Knoop, I owe a lot to Prof. Anton Ertl. Not only did his work start my interest in interpreters and their optimization, he consistently provided valuable information concerning related work, too. Furthermore, he offered helpful comments on several issues and ideas as well as supported the previously mentioned grant proposal for the Austrian Science Fund (FWF).

Many thanks are due for Prof. Michael Franz, who immediately agreed to be the co-examiner for this thesis. Upon my presentation at the University of California, Irvine, in October 2010, he suggested the use of time lines for improving the audience's grasp of the actual, historically accurate, development of techniques. I think this has been a very valuable advice, since the presentation benefits a lot from the use of these time lines.

Next, I want to take the opportunity to thank Mario Wolczko from Oracle/Sun Labs in Menlo Park, CA. He pointed out that dynamic compilation is not the same as just-in-time compilation—an important distinction that we pay only little attention to most of the time. While I cannot remember the names of all participants there, I want to thank them for their hospitality and nice, informative conversations following my talk.

Furthermore, I want to thank Urs Hölzle and Gilad Bracha for helping me to clarify the origins of the interpretative inline caching technique using interleaved pointers, which Urs Hölzle mentions in his thesis and have been implemented first by Robert Griesemer in 1996.

Then, I want to take the opportunity to thank Roland Lezuo, a fellow graduate student who allowed me to re-run important benchmarks on our PowerPC 970 system. The data obtained by these tests contain valuable information and indeed provide some of the best results for our last optimization technique.

My good friend and former colleague Michael Zwick deserves earnest appreciation for reviewing an early draft of this thesis. I consider it a privilege of having met him in Hagenberg. We had a lot of very interesting discussions concerning not only computer science but rather the world at large, and jointly decided to pursue a PhD at the University of Linz in the summer term of 2005.

Without the travel grants of both, the SIGPLAN Professional Activities Committee and the SIGAPP Student Travel Award Programme, I would not have been able to either go to DLS'10 in Reno, Nevada, or SAC'10 in Sierre, Switzerland—this help is very much appreciated.

Finally, I want to thank my family and friends who had to suffer through the misfortune (or fortune, depending on the point of view) of limited exposure to myself during the course of my PhD studies in Vienna. My fiancee, Catherine, has been vital to ensure my mental stability in all non-research related matters—we have had, still enjoy and hopefully will always be having, a great time together. I dedicate this thesis to Catherine, my great family and dear friends in fond remembrance of beloved lost ones: Herbert Brunthaler, Otto Breg, and Michael Bouvier.

Contents

Kurzfassung i

Abstract ii

Acknowledgments iv

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Interpretation . . . 3
  - 1.1.1 Architecture of an Interpreter . . . 3

2 Background 6
- 2.1 Structure and Performance of Efficient Interpreters . . . 6
- 2.2 Interpreter Abstraction-Level . . . 7
- 2.3 Design Decisions for Smalltalk-80 Interpreters . . . 9
- 2.4 Dynamic Compilation . . . 11
- 2.5 Summary . . . 12

3 Purely Interpretative Optimizations 13
- 3.1 Instruction Format . . . 13
  - 3.1.1 Instruction En-/Decoding . . . 13
  - 3.1.2 Data Object Inlining . . . 17
- 3.2 Profiling . . . 20
  - 3.2.1 Using Two Dispatch Routines . . . 21
  - 3.2.2 Swapping the Current Execution . . . 22
- 3.3 Inline Caching . . . 26
  - 3.3.1 Dynamic Typing and its Locality . . . 26
  - 3.3.2 Look-up Caches . . . 28
  - 3.3.3 Interleaving Inline Cache Pointers . . . 29
  - 3.3.4 Quickening . . . 33
  - 3.3.5 Inline Caching Applications . . . 36
- 3.4 Reference Counting . . . 40
  - 3.4.1 Interpreter Operations Causing Reference Count Operations 41
  - 3.4.2 Simple Abstract Interpreter . . . 42
  - 3.4.3 Quickening . . . 45
- 3.5 Partial Stack Frame Caching . . . 47

3.5.1 Basic Idea . . . 47
3.5.2 Allocating Stack Frame Cache Slots . . . 48
3.6 Interpreter Instruction Scheduling . . . 53
3.6.1 Formalization . . . 54
3.6.2 Finding Computational Kernels . . . 55
3.6.3 Scheduling Instructions . . . 58
3.6.4 Compiling Instruction Schedules . . . 63
3.7 Code Generator . . . 64
3.7.1 Architecture . . . 67
3.7.2 Implementation in Numbers . . . 68

4 Related Work **70**
4.1 Purely Interpretative Inline Caching . . . 70
4.2 Reference Count Quickening . . . 71
4.3 Interpreter Instruction Scheduling . . . 73

5 Evaluation **75**
5.1 System Setup and Configurations . . . 75
5.2 Evaluation of Optimization Potential . . . 76
5.2.1 Dynamic Bytecode Frequencies . . . 76
5.2.2 Analysis of Local Variables . . . 78
5.2.3 Analysis of Function Calls . . . 79
5.3 Analysis of Reference Count Operations . . . 80
5.4 Performance Evaluation . . . 81
5.4.1 Detailed Speedup Factors . . . 81
5.4.2 Results per Optimization Technique . . . 82
5.4.3 Interpreter Instruction Scheduling . . . 84

6 Conclusions **86**
6.1 Summary of Contributions . . . 86
6.2 Future Work . . . 87
6.3 Interpreter Optimization Recommendations . . . 89

Bibliography **90**

A Detailed Benchmark Evaluation **97**
A.1 Binarytrees . . . 97
A.2 Fannkuch . . . 100
A.3 Fasta . . . 104
A.4 Mandelbrot . . . 107
A.5 Nbody . . . 111
A.6 Spectralnorm . . . 114
A.7 Django . . . 117
A.8 AI . . . 121
A.9 Reference Count Quickening Details . . . 124

B Comparison of Benchmark Results **125**

Curriculum Vitae **131**

List of Figures

2.1 Native machine instructions for interpreter operation implementation. 8

3.1 Standard irregular Python bytecode encoding. 14
3.2 Optimized regular bytecode instruction format. 14
3.3 Example for relocation procedure on 32 and 64 bit systems. . . . 16
3.4 Illustration of constant object inlining. 17
3.5 Comparison of the assembly generated for `INCA_LOAD_CONST` (*left*) and `LOAD_CONST` (*right.*) 18
3.6 Illustration and implementation of global object inlining. 19
3.7 Relocating instruction pointer. 23
3.8 Relocating the stack of `PyTryBlocks`. 24
3.9 Resolving ad-hoc polymorphism. 26
3.10 Inline Cache states. 27
3.11 Interleaving inline cache pointers. 29
3.12 Illustration of inline caching using interleaved inline cache pointers. 30
3.13 Purely interpretative inline caching using interleaved pointers. . . 30
3.14 Quickening in the Java virtual machine. 34
3.15 Purely interpretative inline caching using quickening. 34
3.16 Illustration of immediate reference counting [Ung86]. 40
3.17 Illustration of deferred reference counting [Ung86]. 41
3.18 Redundant reference count operations. 42
3.19 Illustration of redundant reference count operations. 42
3.20 Finding redundant reference count operations using an abstract interpreter. 44
3.21 Elimination of redundant reference count operations by quickening. 45
3.22 Partial stack frame caching illustrated. 48
3.23 Computing the score for local variable occurrences. 49
3.24 Instruction scheduling. 53
3.25 Graph from the instructions of the kernel for `nbody` benchmark. . 60
3.26 Example of the gdb output on the left side, and the corresponding Python data structure definition on the right side. 65
3.27 UML class diagram of instruction hierarchy. 67
3.28 Flow diagram of instruction set generation. 68

4.1 Time line for inline caching. 71
4.2 Time line for reference count quickening. 72

5.1 Distribution of the number of local variables per stack frame. . . 78
5.2 Frequencies of call types grouped by number of arguments and call targets. 79
5.3 Reference count operations per bytecode. 80

A.1 IIS intermediate graph for `binarytrees` benchmark. 99
A.2 IIS intermediate graph for `fannkuch` benchmark. 102
A.3 IIS intermediate graph for `fasta` benchmark. 106
A.4 IIS intermediate graph for `mandelbrot` benchmark. 109
A.5 IIS intermediate graph for `nbody` benchmark. 112
A.6 IIS intermediate graph for `spectralnorm` benchmark. 116
A.7 IIS intermediate graph for `django` benchmark (*without* edges of weight ≤ 2). 119
A.8 IIS intermediate graph for `ai` benchmark (*without* edges of weight ≤ 1). 122

B.1 Benchmark run-times per optimization technique. 126
B.2 Benchmark run-times for different interpreter instruction schedules. 127
B.3 Benchmark run-times per optimization technique. 128
B.4 Benchmark run-times for different interpreter instruction schedules. 129
B.5 Benchmark run-times for different interpreter instruction schedules. 130

List of Tables

2.1 Reference of reported speedup factors for several techniques [Bru09]. 7

3.1 Dynamic frequencies of instruction types. 15
3.2 Distribution of instruction types using the irregular instruction set. 20
3.3 Optimized `CALL_FUNCTION` derivative instructions. 37
3.4 Dynamic bytecode frequency for `genRandom` function of benchmark program `fasta`. 57
3.5 Dynamic bytecode frequency for an anonymous list comprehension of benchmark program `fasta`. 57
3.6 Dynamic bytecode frequencies for kernel in `advance`. 59
3.7 Interpreter Instruction Schedule for the `nbody` benchmark. 62
3.8 Maximum number of operation implementation fix-ups per benchmark. 64
3.9 Types with context-dependent functions. 65
3.10 Break-down of instructions generated. 69

5.1 Overall comparative dynamic instruction frequency. 76
5.2 Total coverage of calls covered per number of local variables. . . 78
5.3 Number of reference count operations per benchmark. 80
5.4 Speedup factors per benchmark for all interpreter configurations on the Intel Nehalem i7-920. 81
5.5 Speedup factors per benchmark for all interpreter configurations on the PowerPC 970. 82
5.6 IIS Speedup factors per benchmark on the Intel Atom N270. . . 83

A.1 Comparative dynamic instruction frequency for the `binarytrees` benchmark (Argument: 14). 97
A.2 Instruction trace and frequency for `make_tree` function of `binarytrees` benchmark. 98
A.3 Instruction trace and frequency for `check_tree` function of `binarytrees` benchmark. 98
A.4 Computed interpreter instruction schedule for the `binarytrees` benchmark (Argument: 14). 100
A.5 Comparative dynamic instruction frequency for the `fannkuch` benchmark (Argument: 9). 100
A.6 Instruction trace and frequency for `fannkuch` function of `fannkuch` benchmark. 101

A.7 Computed interpreter instruction schedule for the `fannkuch` benchmark (Argument: 9). 103
A.8 Comparative dynamic instruction frequency for the `fasta` benchmark. (Argument: 50,000) 104
A.9 Instruction trace and frequency for `genRandom` function of `fasta` benchmark. 104
A.10 Instruction trace and frequency for the anonymous list comprehension of the `fasta` benchmark. 105
A.11 Computed interpreter instruction schedule for the `fasta` benchmark (Argument: 50,000). 107
A.12 Comparative dynamic instruction frequency for the `mandelbrot` benchmark. (Argument: 500) 107
A.13 Instruction trace and frequency for the `mandelbrot` function of the `mandelbrot` benchmark. 108
A.14 Computed interpreter instruction schedule for the `mandelbrot` benchmark (Argument: 500). 110
A.15 Comparative dynamic instruction frequency for the `nbody` benchmark. (Argument: 50,000) 111
A.16 Computed interpreter instruction schedule for the `nbody` benchmark (Argument: 50,000). 113
A.17 Comparative dynamic instruction frequency for the `spectralnorm` benchmark. (Argument: 400) 114
A.18 Instruction trace and frequency for the `eval_A` function of the `spectralnorm` benchmark. 114
A.19 Instruction trace and frequency for the `eval_A_times_u` function of the `spectralnorm` benchmark. 115
A.20 Computed interpreter instruction schedule for the `spectralnorm` benchmark (Argument: 400). 117
A.21 Comparative dynamic instruction frequency for the `django` benchmark. 117
A.22 Instruction trace and frequency for the `force_unicode` function of the `django` benchmark. 118
A.23 Computed interpreter instruction schedule for the `django` benchmark. 120
A.24 Comparative dynamic instruction frequency for the `ai` benchmark. 121
A.25 Instruction trace and frequency for the `n_queens` function of the `ai` benchmark. 121
A.26 Computed interpreter instruction schedule for the `ai` benchmark. 123
A.27 Relative reduction of reference count operations per benchmark. 124
A.28 Number of reference count operations per bytecode per benchmark. 124

Chapter 1

Introduction

Improving the performance of virtual machines has been a topic of considerable interest during the past 25 years, and continues to be an active research area until today. Its importance can hardly be neglected and it is indeed difficult to imagine a world without high performance implementations of Java and C#. However, most of the research has focused almost exclusively on improving the performance of dynamic compilation sub-systems. This neglects the fact that many important languages use interpreters without these dynamic compilation sub-systems. These languages are powering much of the Internet infrastructure, and indeed have been doing so from the early beginnings in the mid-90s, to today's modern Web 2.0 offering personalization capabilities, also known as the social web. Perl has been the language of choice of the mid-90s Internet, and continues to enjoy high popularity. Currently, Python and Ruby seem to have captured most of the market—and more importantly—mind share for programming the server-side of web applications. The same market- and mind-share is less fragmented on the client side: here the undefeated champion is JavaScript. Consequently, based on these experiences, the likelihood of those systems powering the next iteration of Internet innovation—the Web 3.0—is high.

Most of the advances in dynamic compilation sub-systems can be traced back to optimizing the performance of processing high-level programming languages, viz., Smalltalk and Self. Interestingly, the most successful descendants of this technology are the high-performance just-in-time compilers for Java and C#, which—in comparison to the programming languages powered by their ancestors—are much more low-level, e.g., statically typed instead of dynamically typed programming languages, and having primitive data types instead of the everything-is-an-object paradigm. The spiritual descendants of Smalltalk and Self, such as Perl, Python, and Ruby, however, have not kept up with their ancestors' achievements. The notable exception to this observation is the case of JavaScript, which quite recently has seen a surge in interest for high performance execution (cf. [GES+09]).

Often, the primary reason for lacking a dynamic compilation sub-system is a lack of resources. Particularly in the early stages, many projects—often having just one implementer—cannot afford to allocate valuable resources for the design and implementation of a dynamic compilation sub-system—per se a non-trivial artifact that significantly affects the complexity of any programming language

implementation. These problems aside, a dynamic compilation sub-system sacrifices two of the most important benefits of implementing an interpreter:

- ease-of-implementation, and
- portability.

The key advantage of dynamic compilation sub-systems is performance. Hence, when optimizing an interpreter, one usually faces the dilemma of prioritizing these conflicting properties. That is where purely interpretative optimizations enter the arena. They enable to improve the performance of an interpreter *without* sacrificing its innate characteristics. In comparison to research in dynamic compilation, the area of purely interpretative optimizations has gotten significantly less attention from the research community. This is unfortunate, since this area offers two desirable properties.

First of all, the area is interesting, because often simple techniques have a big impact. While computer science usually considers simplicity to be a favorable trait by itself, in the context of interpreters this has the added benefit of keeping implementation times minimal. In fact, implementing the purely interpretative optimizations presented in this dissertation is a matter of days, or weeks at most and yields considerable, yet portable, performance improvements. This compares favorably to the effort that is necessary for implementing a dynamic compilation sub-system, which usually is a multi-year team-effort. The notable exceptions being the just-in-time compiler for Lua by Mike Pall (LuaJIT) and the just-in-time compiler of the Squeak Smalltalk project, created by Eliot Miranda (CogVM).

Second, research in the area of purely interpretative optimizations is important, because it offers programming language implementers viable options to significantly increase their run-time performance while keeping the benefits of the interpreters.

Recent research in dynamic compilation sub-systems addresses the issue of implementation effort, too, which confirms its importance. The basic idea is to use an existing virtual machine infrastructure with a dynamic compilation sub-system as a basis for another interpreter that runs on-top of it ([YWF09, BCFR09].)

We recognize the importance and general applications of using the quickening optimization to perform dynamic optimizations based on available run-time information. We present novel techniques using quickening to achieve very efficient purely interpretative inline caching (cf. Section 3.3), and to eliminate substantial amounts of reference count operations in immediate reference counting (cf. Section 3.4). Furthermore, we introduce a technique to optimize frequently occurring load-store instructions of stack-based interpreter architectures (cf. Section 3.5). Finally, we show how to use a derivative of instruction scheduling to optimize hardware instruction-cache utilization on the native machine processor executing the interpreter (cf. Section 3.6).

Using these techniques gives us speedups of up to 2.4176 on a modern Intel i7 Nehalem architecture (cf. Section 5.) In addition to the techniques themselves, we present details of the code generator we use to generate customized versions of the Python interpreter (cf. Section 3.7.) Finally, we provide a set of recommendations for interpreter implementers with the purpose of giving practical, hands-on, advice to jump-start adoption of the techniques presented herein (cf. Section 6.3.)

1.1 Interpretation

A compiler is a program that reads a program p in some source language S and translates it to a destination language D. Whenever the destination language D is a native machine language, this means that the program p can be executed on a native machine that knows how to process D—without the compiler being present at all. This means, for example, that we can compile the program p to some native machine representation different from the one that runs the compiler. In consequence, the compiled program can then be executed on the target hardware.

Contrary to this approach, an interpreter is a program that runs itself while interpreting a program p. Hence, it does not translate the program p from S to D, but directly executes the statements comprising p. Therefore, the interpreter must know how to process all statements of p. If it does not, either the input program is incorrect, or some run-time error occurred. We call the language of p the *host language*—this is the language the interpreter processes. The interpreter itself, however, is usually written in another language that we call the *interpreter-* or *implementation language*. When the host language and implementation language of an interpreter are identical, we speak of a *meta-circular interpreter*.

1.1.1 Architecture of an Interpreter

In order for an interpreter to compute something, it needs to read an input program p, which contains statements of the host language. Therefore, it needs to parse the program p using the grammar of the host language. Using this grammar, the interpreter constructs an abstract syntax tree—or AST for short—which reflects the program's structure. Now, an interpreter implementation can either:

1. start executing the program p using its AST, or
2. translate the program p to a more efficiently interpretable representation.

In the first case, the AST interpreter needs to have knowledge on how to process every node of the AST. In the second case, the interpreter translates the program p to an intermediate representation known as *bytecode*, or (*instruction-*) *opcode*. Now, an interpreter processes sequences of bytecodes instead of AST nodes. This class of interpreters is commonly referred to as *virtual machines*. Similar to AST interpreters that need to know how to interpret every node of the AST, a virtual machine needs to have an *instruction set* that formally specifies which instruction-opcodes it understands.

The compilation from the host language source code of the input program to the bytecodes of the virtual machine can either be *ahead-of-time*, as is the case for the Java programming language, or can happen *just-in-time*, i.e., directly before interpreting the code, which is the way, for example, the Python interpreter works.

Usually, bytecode interpreters or virtual machines are regarded to be more efficient than AST interpreters, because the implementation of their instruction sets is co-located in instruction memory, i.e., they considerably improve their locality of reference in comparison to AST interpreters. Interestingly, while the

term bytecode conveys that the instruction-opcodes have a size of one byte—which they frequently do—an interpreter can have different instruction formats, such as a word-sized instruction format, too.

Whenever a virtual machine completes the execution of one instruction, it needs to decode the following instruction and transfer the control to its successor. We call this the *instruction dispatch*, which subsumes the *instruction decode* phase. Therefore, any bytecode interpreter must maintain a distinct *instruction pointer* or *program counter* to implement control flow. An AST interpreter does not require an instruction pointer, since the AST contains explicit control flow nodes, such as nodes describing if-then-else statements.

In an AST interpreter, the *operands* of an operation, such as an addition with numerical operands, are child nodes of the corresponding sub-tree of the addition operation AST node. In contrast to this, the virtual machine interpreter needs to pass operands or *instruction arguments* using one of the following methods:

- Stack architecture: all operands are pushed onto and popped off a data stack.
- Register architecture: every instruction includes the references identifying registers holding the operands.

Interpreters with a stack architecture have been the predominant implementation choice, and require an additional *stack pointer* to implement data flow. This is primarily due to compactness reasons: it is commonly suggested that bytecodes for the stack architecture do not need to carry the register information for its operands and therefore require less space. However, recently implemented virtual machines trade the additional space for performance reasons: An interpreter using a register architecture requires fewer instructions for loading and storing operands. Among the interpreters using a register-architecture are the Lua interpreter, Google's DalvikVM of the Android project, and the Parrot virtual machine for executing Perl.

Starting with one of the very first wide-spread interpreters, Lisp 1.5 [McC62], interpreters most often offer *automatic memory management*. All flavors of automatic memory management techniques are present in various virtual machines, and research in the area of efficient garbage collection techniques is highly interrelated with research in the area of efficient execution of interpreters. It is, however, not a necessity for virtual machines to have automatic memory management, i.e., one could very well design an interpreter that leaves memory management to the host language.

Another frequently encountered architectural feature of interpreters is in the area of type systems. Since an interpreter executes the instructions of an input program, the actual *operation implementation* has the ultimate information at its disposal. Dynamically typed programming languages rely on this feature, i.e., depending on the actual operand types, proper operations are selected at run-time. In general, an interpreter implements dynamic typing by having type-generic instructions. For example, JavaScript, Lisp, Perl, Python, and Ruby interpreters feature dynamic typing. While many interpreters use dynamic typing, they can support static typing as well, e.g., the Java virtual machine, or the Pascal virtual machine that processes p-code. In those virtual machine interpreters, all bytecodes are type-specific instead of type-generic: For example, compare the Java virtual machine's `iadd` instruction for adding integers to

the untyped `BINARY_ADD` instruction of Python. Another architectural concern within the implementation of type systems in interpreters is to decide whether the typing rules are strongly or weakly enforced. Though a Python program is dynamically typed, the interpreter does not perform operations on incompatible operands, e.g., adding a string and an integer results in a type error, i.e., the Python interpreter is strongly typed. Contrary to this example, the Perl and Lua interpreters are weakly typed, e.g., when invoking the addition operator on a string and an integer, their interpreters will try to coerce the string into a meaningful numeric value and proceed with arithmetical addition if successful, i.e., their interpreters will at least try to make a best effort before reporting a type error. In other words, this strong and weak typing discipline can be seen as giving either the operand types or the operation a higher precedence.

A final feature of interpreters is that because of their dynamic nature, they can evaluate arbitrary code at run-time. Many interpreters choose to offer this feature to the host language, which among other things allows for dynamically loading and processing code without restarting the interpreter.

Summing up, there are many design alternatives determining the final architecture of an interpreter. Usually, interpreters can be fairly well classified using the following characteristics:

- AST-interpreter or virtual machine interpreter,
- stack or register architecture instruction format,
- memory management technique,
- static or dynamic typing of host language objects,
- strongly or weakly enforced typing host language expressions.

Chapter 2

Background

2.1 Structure and Performance of Efficient Interpreters

In 2003, Ertl and Gregg published "The Structure and Performance of Efficient Interpreters" [EG03b], which discusses many interesting points—so many in fact that we decided to name this section after the paper. Originally, the paper was a response to an earlier paper by Romer et al. [RLV+96], which analyzed the effect of hardware for various interpreters (MIPSI, Java, Perl, Tcl). Romer et al. conclude that interpreters would not benefit very much from dedicated hardware. Ertl and Gregg show, however, that this is actually not true, since interpreters perform exceptionally high amounts of indirect branch instructions. Subsequently, studying the performance of various branch predictors, Ertl and Gregg report a speedup factor of up to 4.77 between no branch prediction and a good branch predictor. We are, however, not directly interested in this paper's results of branch prediction. More importantly, this paper contains valuable information on interpreters and interpretation in general.

First of all, Ertl and Gregg relate interpreter efficiency to optimizing native code compilers, and find that while efficient interpreters perform with a slowdown by a factor of 10 when compared to an optimizing native code compiler, inefficient interpreters have a slowdown by a factor of 1000. Consequently, the relative-slowdown between efficient and inefficient interpreters is ten-fold higher than the slowdown between efficient interpreters and optimizing native code compilers. Hence, optimization of inefficient interpreters has a disproportionally higher amount of speedup potential.

Next, Ertl and Gregg analyze the performance of the following interpreters: Gforth, OCaml, Scheme48, Yap, Perl, Xlisp. While Gforth, OCaml, Scheme48 and Yap are categorized as efficient interpreters, Perl and Xlisp benchmarks are used for comparison purposes as inefficient interpreters. The working hypothesis is that because operation implementation for many interpreter instructions is usually small, instruction dispatch constitutes the most expensive part of an interpreter, since it requires at least one expensive indirect branch instruction. Therefore, optimization techniques focusing on minimizing the overhead in instruct dispatch have substantial speedup potential (cf. Table 2.1.)

Optimization Technique	Speedup Factor	Reference
Threaded Code (compared to switch dispatch interpreter)	up to 2.02	[EG03b]
Superinstructions (compared to threaded code interpreter)	up to 2.45	[EG03a]
Replication + Superinstructions (compared to threaded code interpreter)	up to 3.17	[EG03a]
Register vs. Stack Architecture (both using switch dispatch)	1.323 avg	[SCEG08]
Register vs. Stack Architecture (both using threaded code)	1.265 avg	[SCEG08]

Table 2.1: Reference of reported speedup factors for several techniques [Bru09].

Interesting facts are provided concerning the performance of inefficient interpreters:

> The behavior of Xlisp and Perl is very different from the other interpreters. Xlisp has a low misprediction rate for all predictors. We examined the code and found that most dynamically executed indirect branches do not choose the next operation to execute, but are switches over the type tags on objects. Most objects are the same type, so the switches are quite predictable. The misprediction rates for Perl are more in line with other interpreters, but figure Fig. 7 shows that improving the prediction accuracy has little effect on Perl. Non-return indirect branches are simply too rare (only 0.7%) to have much effect.

Therefore, in the Xlisp case the original hypothesis is violated—instruction dispatch does not cause the most indirect branches, but dynamic typing does. In the Perl case, the reason for lacking success might be attributed to the fact that it does not in fact use a bytecode interpreter, but its architecture is more akin to a highly optimized AST interpreter.

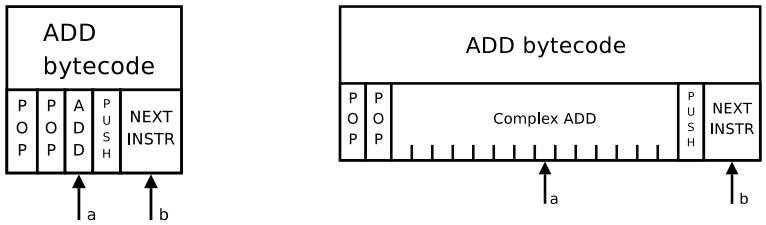
In 2004, Vitale and Abdelrahman [VA04] reported their results on bringing the quite successful optimization of Piumarta and Riccardi [PR98] to the Tcl interpreter. Interestingly, they reported not only that their speedups were far lower than expected (cf. Table 2.1), but there were some cases were they actually found slowdowns.

2.2 Interpreter Abstraction-Level

Because of the result on inefficient interpreters by Ertl and Gregg and surprising result of Vitale and Abdelrahman (cf. Section 2.1), we decide to investigate the performance of interpreters more deeply. Note that parts of this section have been published before [Bru09].

The hypothesis of Ertl and Gregg [EG03b] was that the operation implementation of most interpreter instructions is small and therefore the instruction dispatch overhead is considerable. Therefore, we analyze several interpreter implementations, viz. Java, OCaml, Python, and Lua to verify this hypothesis.

Whereas we find that this hypothesis holds for the first two interpreter implementations, Java and OCaml, our investigation of Python and Lua shows that their operation implementation is usually much more complicated.



(a) Operation implementation realized by native machine.

(b) Expensive operation implementation.

Figure 2.1: Native machine instructions for interpreter operation implementation.

Figure 2.1a shows how Java, OCaml and many other interpreters, such as the Gforth interpreter delegate the actual operation implementation to the executing native machine. Contrary to these interpreters, Figure 2.1b illustrates that the operation implementation for Python, Lua, and interpreters for other programming languages, such as JavaScript, Perl and Ruby, is much more complex and cannot be delegated to the native machine. In consequence, optimizations targeting instruction dispatch are particularly effective on interpreters where the ratio of native machine instructions in operation implementation a[1] to instruction dispatch b is smaller than 1 ($\frac{\mathsf{a}}{\mathsf{b}} < 1$, cf. Figure 2.1a.) Even more so, when the instruction dispatch contains expensive instructions such as indirect branches. For example, the regular switch-based instruction dispatch technique requires 9-10 native machine instructions, whereas the optimized direct threaded-code instruction dispatch requires only 3-4 instructions [EG01]. Besides halving the native machine instructions necessary for instruction dispatch, direct threaded code eliminates an additional indirect branch instruction, too. Since the native machine realizes most of the operation implementations of interpreter instructions, we call this class of interpreters *low abstraction-level virtual machines*.

If the ratio of operation implementation a to instruction dispatch b is much greater than 1 ($\frac{\mathsf{a}}{\mathsf{b}} >> 1$, cf. Figure 2.1b), the optimization potential of optimizations targeting instruction dispatch decreases. Usually, the reason for this is that the operation implementation itself is rather expensive, having many branches and function calls. Because the native machine cannot provide the necessary functionality, we call this class of interpreters *high abstraction-level interpreters*.

The classification of interpreters with respect to their abstraction level explains the varying optimization potential of optimization techniques targeting instruction dispatch—which explains the result of Vitale and Abdelrahman [VA04]. Interestingly, Piumarta and Riccardi anticipated this in the conclusion of their paper [PR98]:

> The expected benefits of our technique are related to the average semantic content of a bytecode. We would expect languages such

[1] We use the sans-serif font for establishing the relationship to Figure 2.1

> as Tcl and Perl, which have relatively high-level opcodes, to benefit less from macroization. Interpreters with a more RISC-like opcode set will benefit more — since the cost of dispatch is more significant when compared to the cost of executing the body of each bytecode.

Consequently, to successfully optimize high abstraction-level interpreters, we have to focus our efforts on optimizing the operation implementation instead. We need to inspect the operation implementation of high abstraction-level interpreters to answer the question of why it cannot be delegated to the native machine. All of the previously mentioned programming languages having high abstraction-level interpreters are highly abstract programming languages, too. They share the following features:

- dynamic typing,
- automatic memory management,
- lack of primitive data types.

Therefore, the operation implementation for many of the interpreter's instructions needs to:

- dynamically resolve the types of operands and select the proper operation implementation for the given types,
- manage the memory using one of the corresponding techniques,
- box and unbox primitive data from objects.

For example, to just add two integers, a high abstraction-level virtual machine addition instruction must first determine the types of both operands, select the actual integer addition operation based on the operand types, unbox the integer values from the operand objects, perform the addition and box the result again before pushing it onto the stack again. If the interpreter uses reference counting [Col60] for automatic memory management, it has to decrement the reference counts for both operands and increment the reference count for the object holding the result value, too.

A brief look at the history of programming languages gives important insights as to which optimization techniques reduce the overhead in operation implementation. The Smalltalk [GR83] and Self [US07] programming languages come to mind—both of which constitute high abstraction-level virtual machines that have seen considerable performance improvements using pioneering optimization techniques.

2.3 Design Decisions for Smalltalk-80 Interpreters

In 1982, Allen Wirfs-Brock published an important article detailing several degrees of freedom and trade-offs one faces when implementing a Smalltalk interpreter [WB82]. In 1983, this article has been published along others in a very important book for high abstraction-level virtual machine implementers, "Smalltalk-80: Bits of History, Words of Advice" by Glenn Krasner.

The formal specification of Smalltalk-80 [GR83] defines a virtual machine architecture with an instruction set, primitive data types, and automatic memory management. Allen Wirfs-Brock mentions the following points as design decisions:

1. *Host Processor*: Here we get advice on possible CPU speed, main memory ("at least 1 megabyte of main memory") requirements, number of available registers, etc. This section is probably least relevant for nowadays modern CPUs.

2. *Implementation Language*: The advice on choice of a programming language is very important. It is recommended to implement an interpreter first in a high-level programming language to get a "feel" for the language and a correct implementation first. For ultimate performance, Allen Wirfs-Brock recommends to implement the interpreter in a low-level language, i.e., assembly language, mentioning that their Smalltalk implementation for a Motorola 68000 needs approximately 5,000 native machine instructions.

3. *Object Pointer Formats*: A section detailing pro and contra arguments concerning specific tag-bit positions for supporting tagged integers as a primitive data type in combination with regular object pointers.

4. *Object Memory*: Discusses several concerns when implementing an object table. Modern implementations on modern CPUs need not necessarily have an object table, i.e., the points therein do not actually correspond to our discussion.

5. *Bytecode Interpreter*: Mentions the use of the indirect threaded code instruction dispatch optimization technique. Other discussion points include the amount of caller/callee save registers for frequent function calls with only few bytecodes (10 instructions or less).

6. *Memory Management*: Aside of mentioning that their initial interpreter spent 70% of its time in memory management routines, this point covers Smalltalk specifics for object allocation and storage reclamation. Object allocation in Smalltalk is expensive because every function call needs to allocate stack frame ("Context") objects—these are visible to the host-language. In storage reclamation, they observe that switching from immediate reference counting [Col60] to deferred reference counting [DB76] is beneficial. Note that this discussion dismisses tracing garbage collectors for taking too much pause time; this was also before the implementation of generation scavenging by Ungar in 1984 [Ung84].

While points 1 and 4 are probably not an issue today anymore, the other points contain valuable information. For instance, the choice of implementation language even today substantially influences performance. Even though in 1987 Eliot Miranda presented a highly portable C-based Smalltalk implementation called BrouHaHa [Mir87], he had to resort to do some sort of manual global register allocation and used the same order of variable declaration to influence the code generated by the C compiler. In early 2010, Mike Pall, the author of the LuaJIT just-in-time compiler for the Lua programming language posted this on the Internet [Pal10]:

> It's much easier to record what an interpreter is doing. Just patch its dispatch table and intercept every instruction.
>
> The *small gains of a simplistic compiler over a carefully hand-optimized interpreter are just not worth the trouble (the LJ1 JIT compiler is not much faster than the LJ2 interpreter, sometimes it's worse)*[2]. Better focus on improving the trace compiler and stay 'on-trace' as much as possible.
>
> Another subtle point, often forgotten in the reports about Mozilla's plan with JägerMonkey: the method compiler only triggers trace recording, but they still need the interpreter to actually record the traces! So now they gotta keep three engines in sync (interpreter, method compiler, trace compiler). I'll leave it to your judgment whether that's a smart move or pure desperation.

The mentioned LuaJIT-2 interpreter is an interpreter written in assembly language, which gives the implementer the ultimate control over machine details, such as register usage, etc. This eliminates a level of indirection when coding in C and using its compiler for compiling the interpreter. The same approach was used by the implementers of the Strongtalk interpreter in 1996.

Design point no. 3, *Object Pointer Formats*, helps in eliminating the overhead of (un-)boxing objects. Point no. 5, *Bytecode Interpreter*, includes advice on optimizing the costs incurred by instruction dispatch. Finally, the last point, *Memory Management*, recommends to spend a considerable amount of time thinking about which automatic memory management technique to use. There is no commonly accepted recommendation for using technique x in any implementation, since the performance is highly dependent on the application scenario and on external design considerations, such as memory consumption, pause times, etc. We refer the interested reader to the canonical reference of Jones and Lins [JL96], plus an updated survey article by Jones [Jon07].

2.4 Dynamic Compilation

Together, the recommendations of Section 2.3 allow us to attack at least some of the problems of expensive operation implementation mentioned earlier in Section 2.2, the notable exception being dynamic typing. The major innovation here came in 1984, with the landmark publication of "The Efficient Implementation of the Smalltalk-80 System" by L. Peter Deutsch and Allan Schiffman [DS84]. This paper introduced several important ideas:

- dynamic compilation from virtual machine code to native machine code,
- inline caching,
- de-optimization,
- throwing away code and re-generating it instead of paging-in cached code from memory.

[2] Emphasis added by the author.

These ideas have been very successful and have subsequently led to important research on optimizing the SELF programming language. The PhD-theses of Craig Chambers [Cha92] and Urs Hölzle [Höl94] give substantial details to numerous optimizations, such as customization, and polymorphic inline caching. In 2003, Aycock published a good article giving a historical perspective to just-in-time compilation [Ayc03]. In 2008, Kotzman et al. published an article detailing the design of the Java HotSpot™ client compiler [KWM+08]. In 2009, Gal et al. [GES+09] presented their work on using trace-based compilation for speeding up the TraceMonkey JavaScript engine of the Mozilla Firefox browser. This work changes the compilation unit of the dynamic compilation sub-system: usually, whole methods were compiled, in a trace-based setting, however, traces of instructions are compiled. These traces collect instructions across functions, i.e., they perform function-inlining implicitly.

Dynamic compilation systems achieve premium performance, often at the expense of additional memory—using the well-know optimization principle of trading space for time. Independent of the programming language they optimize, one can categorize dynamic compilation sub-systems according to their compilation time:

- Mixed execution: An interpreter starts executing the bytecode instructions and collects profiling and type information. This information is subsequently used in the dynamic compilation step.
- Just-in-time compilation: No interpreter is used, but a simple and fast base compiler generates native code just-in-time before transferring the control to the compiled method. This base compiler adds collectors for profiling information. Once these information conclusively identifies a frequently used piece of code, an optimizing compiler re-compiles it.

Inline caching effectively eliminates the overhead introduced by dynamic typing. Therefore, it represents the missing piece for optimizing operation implementation of interpreters.

2.5 Summary

This chapter introduced the notion of virtual machine abstraction-levels as the primary reason for the varying optimization potential of optimization techniques focusing on instruction dispatch. Using this observation, we briefly explored the historical achievements on similar high abstraction-level interpreters, viz. the Smalltalk and SELF systems. In consequence, this dissertation recognizes the following features to substantially affect the performance of their interpreters, as well as proposes optimization techniques to increase efficiency:

- Optimize dynamic typing by using purely interpretative inline caching.
- Optimize reference counting by eliminating redundant reference count operations.

The next chapter presents additional optimizations, however, all but one of them (interpreter instruction scheduling, cf. Section 3.6) derive their existence from these observations, viz. to minimize the overhead in operation implementation of interpreters.

Chapter 3

Purely Interpretative Optimizations

This chapter describes the implementation of our optimization techniques, as well as the rationale behind them. The primary vehicle for demonstrating all of these implementations is the Python 3.x series interpreter, specifically, the Python 3.1 version. Our choice for the Python interpreter is that it has a simple and clean code base that is a good representative for many other interpreter implementations. We present optimization techniques that increase efficiency by:

1. Changing the instruction format,
2. Introducing purely interpretative inline caching,
3. Elimination of redundant reference count operations,
4. Optimizing load and store instructions by partial stack-frame caching,
5. Optimizing instruction cache utilization by interpreter instruction scheduling.

3.1 Instruction Format

This section deals with our changes to the instruction format of the Python 3.x series interpreter. Like many other interpreter architectures, the Python 3.x series interpreters use an irregular instruction format. Similar to the instruction format of CPUs, an irregular instruction format requires complex instruction de- and encoding.

3.1.1 Instruction En-/Decoding

We observe the irregularity of the Python instruction format by noticing that not all bytes in the list of bytecodes are actually instruction opcodes. Rather, if an instruction requires an argument, the two adjacent consecutive bytes contain its value (cf. `LOAD_FAST` in Figure 3.1.) In consequence, whenever we decode an instruction, we have to check whether it has an argument/operand that needs to

be decoded, too. Otherwise, the instruction pointer of the interpreter would not be able to properly dispatch to the next instruction.

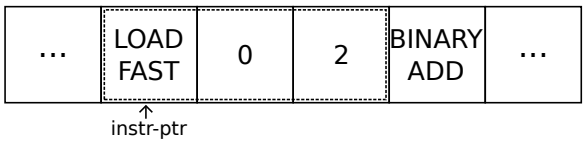


Figure 3.1: Standard irregular Python bytecode encoding.

Since the interpreter needs to decode every instruction it executes, this decoding should be as fast as possible, similar to the importance of optimizing the actual instruction dispatch using some of the previously mentioned optimization techniques, such as threaded code (cf. Table 2.1). The current instruction decoding translates to the following assembly sequence on an x86-64 architecture, as compiled by `gcc` 4.4.3 :

```
1   movzbl  0x0(%r13),      %r15d   ; r13 contains the instruction pointer,
2                                   ; r15 the opcode
3   xor     %r8d,           %r8d    ; zero out oparg, r8
4   add     $0x1,           %r13    ; advance instruction pointer by one
5   cmp     $0x59,          %r15d   ; check if instruction has an argument (i.e., its
6                                   ; value is above 90 (= 0x59)
7   jle     L_SKIP_OPARG_DECODE     ; skip operand decoding if below 90
8   add     $0x2,           %r13    ; advance instruction pointer by two
9   movzbl  -0x1(%r13),     %eax    ; load second byte into eax
10  movzbl  -0x2(%r13),     %r8d    ; load first byte into r8d
11  shl     $0x8,           %eax    ; shift eax by one word to the left
12  lea     (%rax, %r8, 1), %r8d    ; use load-effective-address using offset com-
13                                  ; putation (rax + 1*r8) to calculate oparg (r8d)
14 L_SKIP_OPARG_DECODE:
15  ...
```

We notice that the conditional instruction decoding requires at least a comparison instruction (line 5) and branch (line 7). Whenever we actually have to decode an instruction operand, we need two load instructions to fetch the memory contents (lines 9, and 10 respectively) as well as a shift instruction to prepare the operand value computation using a `lea` instruction. Finally, we need another add instruction to advance the instruction pointer of the interpreter. All in all, this amounts to executing 5 instructions for decoding the instruction operand.

From the development of RISC CPUs, we know that having a regular instruction format enables us to use a fast fixed hardware instruction decoder. Therefore, if we combine the instruction opcode and its operand into one native machine word, we can simplify instruction decoding by quite a bit:

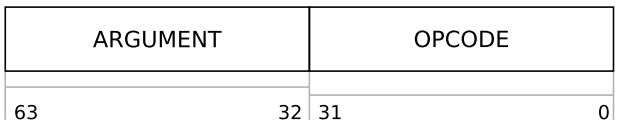


Figure 3.2: Optimized regular bytecode instruction format.

For our target architecture (x86-64), `gcc` 4.4.3 generates the following assembly:

```
1  mov     (%r12), %r8         ; move contents of instruction pointer (r12)
2                              ; location to r8
3  add     $0x8,   %r12        ; advance instruction pointer to next word (8 bytes)
4  mov     %r8d,   0x28(%rsp)  ; store opcode part (r8d) to the stack location of
5                              ; the opcode variable
6  shr     $0x20,  %r8         ; shift the oparg part to the right by 32 bits
```

Using our new, regular, instruction set, we can decode instructions more efficiently. Instead of the 5/10 assembler instructions, we can decode an interpreter instruction using just 4 native machine instructions. The new decoding needs just one memory load (line 1) instead of 3, eliminates the comparison, branch and exclusive-or instruction, and needs just one addition to advance the instruction pointer (line 2), instead of two advances when decoding the instruction operand.

Besides these considerable savings, it is interesting to note that this instruction decoding procedure does not need to allocate a dedicated register to the `opcode` variable (line 1 in the previous listing, `r15d`). Instead, whenever we come to the threaded code instruction dispatch, we can refer to the value of the `opcode` variable by referring to the register holding the instruction pointer (`r12`) with the appropriate size and an empty displacement (`(r12d)`). For less frequent uses of the `opcode` variable, we store its value directly to the stack frame of the interpreter routine (line 4). By removing the conditional operand decoding, the compiler is free to remove all but the addition instruction (line 3) to advance the instruction pointer, if the `opcode` and `oparg` values are not used within the operation implementation, i.e., are dead code.

Note that even though we present this regular instruction format in the context of a 64 bit architecture, it is perfectly applicable to a much more common 32 bit system as well.

Quantitative Analysis

Benchmark	Dynamic Frequency of Instruction Types			
	Absolute		Percentage	
	Argument	No Argument	Argument	No Argument
ai-1	17,965,268	8,928,896	66.80%	33.20%
binarytrees-14	174,540,267	28,521,410	85.95%	14.05%
django-1	35,924,110	5,100,864	87.57%	12.43%
fannkuch-9	61,282,842	15,721,656	79.58%	20.42%
fasta-50000	7,817,146	3,236,110	70.72%	29.28%
mandelbrot-500	78,958,007	14,163,979	84.79%	15.21%
nbody-50000	39,267,950	29,708,384	56.93%	43.07%
spectralnorm-400	134,776,564	70,457,432	65.67%	34.33%
Total	550,532,154	175,838,731	75.79%	24.21%

Table 3.1: Dynamic frequencies of instruction types.

Table 3.1 shows that about three quarters of executed instructions require the more complex irregular operation argument procedure. Therefore, switching

to the regular instruction set eliminates 100% of the comparison and branch instructions of the irregular instruction decoding. In addition, 75% of the operand decoding instructions become unnecessary, thereby saving two memory loads, one addition, and one `lea` instruction.

Relocation

All jumps encoded in the original format include the argument-bytes in their absolute/relative destination positions. Hence, all jumps need to be relocated to match our new instruction encoding.

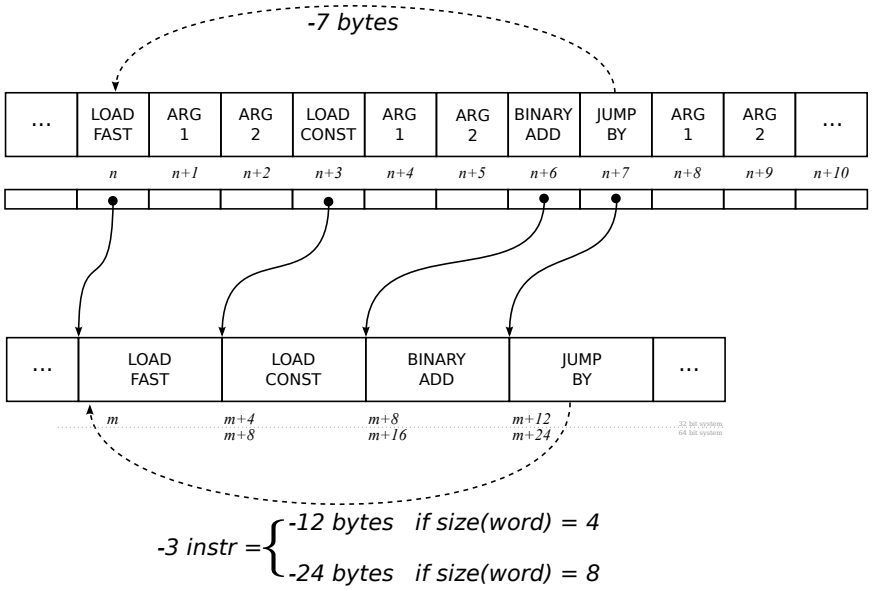


Figure 3.3: Example for relocation procedure on 32 and 64 bit systems.

Algorithm:

- We use the old dispatch mechanism to determine the values of `opcode` and `oparg`. Change a separate pointer *relocate* together with the `next_instr` pointer. The *relocate* pointer points into the second memory area of size n—equal to the number of instructions.
- If we have an instruction that performs a jump, use the following calculation to relocate the jump target: $oparg_{new} := *(relocate + oparg_{old}) - *relocate$ With regard to our concrete example from Figure 3.3, this means:
  - The current instruction pointer is at position $n + 7$, with $oparg_{old}$ having the value -7. The corresponding *relocate* pointer points to $m + 24$.
  - The jump instruction target is at instruction pointer position n. Its corresponding *relocate* pointer (which is located at offset n, too) points to offset m in the new instruction encoding scheme.

– Therefore,
$oparg_{new} := *(relocate + (-7)) - (m + 24)$
$oparg_{new} := (m) - (m + 24)$,
$oparg_{new} := -24$

3.1.2 Data Object Inlining

If we take a closer look at the operand encoding in Figure 3.2, we notice that we reserve full 32 bits for encoding the operand value, whereas the original irregular instruction format only required 16 bits for encoding the same. We notice that even though our interpreter is running on a 64 bit architecture, half of its address space is expressible using the 32 bits that our new regular instruction format provides us with. Consequently, we can inline such data object references directly into the instruction encoding. The following two sections explain two different approaches that use this data object inlining: one for inlining constant object references, the other for inlining global variable references.

Loading Constants

When running the interpreter, we quickly notice that most of the constant objects that our interpreter uses for the computation are allocated in the lower memory area, below 32 bits. This observation makes sense, because many of the used constant objects will already be allocated at start-up time, such as the references to Boolean true and false values in Python, i.e., `Py_True` and `Py_False`.

If we check that the address to a constant data object is in the lower memory area, below 32 bits, we can safely inline its reference in the instruction encoding, replacing the integer array index to that reference.

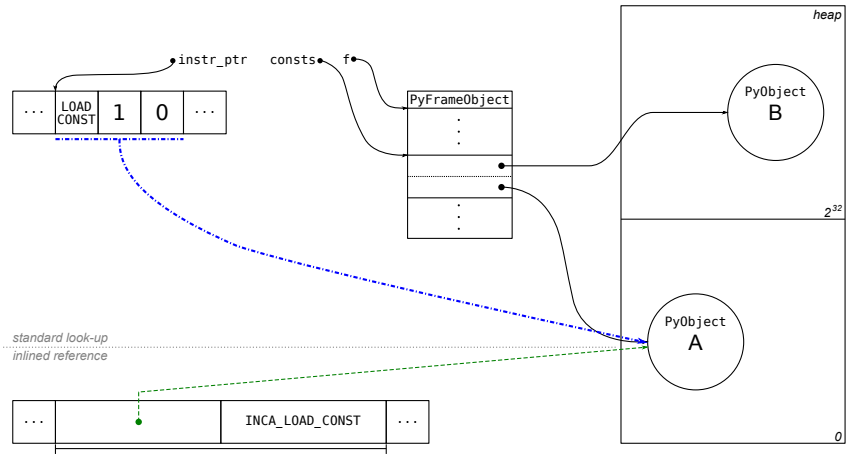


Figure 3.4: Illustration of constant object inlining.

Figure 3.4 shows the effect of using this optimization. We see that the operation implementation can be significantly reduced by using this optimization. Using `gcc` 4.4.3 to compare each of these versions shows that we are able to eliminate 2 native machine instructions that load the memory location pointed to by `consts` pointer:

```
mov     0x0(%rbp),%r13
add     $0x8,%rbp
mov     %r13d,%r15d

sar     $0x20,%r13

mov     %ebp,%eax

addq    $0x1,0x0(%r13)
mov     %r13,(%r12)
add     $0x8,%r12
sub     0x30(%rsp),%eax
mov     %eax,0x78(%rbx)
mov     0x0(%rbp),%eax
mov     0x0(,%rax,8),%rdx
xor     %eax,%eax
jmpq    *%rdx
```

```
mov     0x0(%rbp),%rax
add     $0x8,%rbp
mov     %eax,%r15d
; first load instruction
mov     0xa0(%rsp),%rdx
sar     $0x20,%rax
; second load instruction
mov     0x18(%rdx,%rax,8),%r13
mov     %ebp,%eax

addq    $0x1,0x0(%r13)
mov     %r13,(%r12)
add     $0x8,%r12
sub     0x30(%rsp),%eax
mov     %eax,0x78(%rbx)
mov     0x0(%rbp),%eax
mov     0x0(,%rax,8),%rdx
xor     %eax,%eax
jmpq    *%rdx
```

Figure 3.5: Comparison of the assembly generated for `INCA_LOAD_CONST` (*left*) and `LOAD_CONST` (*right.*)

When we create the new instruction format, we can easily quicken all the occurrences of matching `LOAD_CONST` instructions to the optimized `INCA_LOAD_CONST` instruction derivatives.

Loading Global Variable References

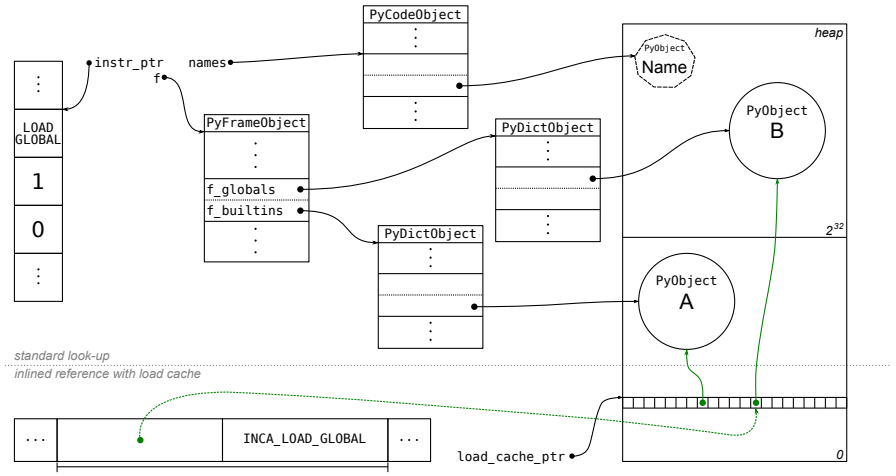
Contrary to the inlining of constant object references described in the previous, references to global data objects are not read-only, i.e., there exist `STORE_GLOBAL` instructions but not `STORE_CONST` instructions. Therefore, straightforward inlining of global data object references would invariably complicate the invalidation mechanism when a store occurs. Moreover, global data object references can occur anytime, whereas we can observe a certain locality of the memory addresses of constant object references being in the lower memory area, below 32 bits. Similar to the concept of hardware cache levels, we therefore introduce a small cache to hold references to global objects.

We allocate an array of n elements of the following data structure in the lower memory area, below 32 bits:

```
typedef struct {
    bytecode_t *ip;
    PyObject *elem;
} load_cache_elem_t;
```

This allows us to present a small set of n references to data objects of the whole memory area by references to `load_cache_elem_t` array elements, which by construction comply with our size restriction of 2^{32} bits. Thus, we can inline any data object reference for an occurrence of a `LOAD_GLOBAL` instruction by inlining a reference to a `load_cache_elem_t` element.

Figure 3.6 shows how this inlining allows us to significantly simplify the operation implementation of `INCA_LOAD_GLOBAL` when compared to a regular `LOAD_GLOBAL` instruction. Instead of the expensive key look-up procedures in




```
TARGET(LOAD_GLOBAL)
    w = GETITEM(names, oparg);
    x = PyDict_GetItem(f->f_globals, w);       /* 1st */
    if (x == NULL) {
        x = PyDict_GetItem(f->f_builtins, w); /* 2nd */
        if (x == NULL)
          load_global_error:
    /* remaining implementation omitted */
```

Figure 3.6: Illustration and implementation of global object inlining.

Python dictionaries, we can just access the inlined load cache element, verify its validity by checking the instruction pointer equivalence and use the actual data object reference.

However, we still need to take care of proper cache invalidation, such that using this optimization does not violate its soundness. Whenever a `STORE_GLOBAL` updates a global data object reference, we need to ensure that any subsequent `LOAD_GLOBAL` instruction that indirectly references that global data object with its inlined `load_cache_elem_t` reference fails. Otherwise, the update of the `STORE_GLOBAL` instruction would not be destructive and the interpreter would continue to use the data object reference it obtains through the inlined `load_cache_elem_t` reference. There are two options available to an implementer: a) a fine grained, and b) a course grained cache invalidation mechanism. The first approach (a) would require us to fetch the currently held data object reference of the global variable slot, i.e., we would need to execute the `LOAD_GLOBAL` logic before actually updating anything. Next, we could search the array of `load_cache_elem_t` records for any references to that object an reset their `ip` field, such that an `INCA_LOAD_GLOBAL` would fail when checking whether the inlined reference still corresponds to its instruction pointer. After successfully ensuring that no `load_cache_elem_t` record references the old data object reference, we can safely update the global variable slot to reference the new data object. Any subsequent `INCA_LOAD_GLOBAL` for such an inlined `load_cache_elem_t` would fail and update itself.

A more course grained approach (b) would be to just invalidate all `load_cache_elem_t` members of the load cache array. All subsequent `INCA_LOAD_GLOBAL` would fail and update themselves. However, we would not have to fetch the old data object reference before updating it in the operation implementation of the `STORE_GLOBAL` instruction. For simplicity, our implementation follows this second, coarse grained, approach.

Finally, we have to describe a small technical issue for successfully implementing the cache miss. When inlining a reference to a `load_cache_elem_t` element, we overwrite the operand value and within our optimized instruction format have no hope of ever retrieving it again. The solution to this is to retrieve its value from the old, irregular, instruction format—which we keep for other purposes, such as debugging, too.

3.2 Profiling

The previous section (Section 3.1) describes in detail how changing the instruction format is a good thing, with respect to optimal performance. However, as we can see in the corresponding figures, changing the instruction format has one important downside: Switching to a regular instruction format considerably increases memory requirements of representing the instructions. In our new regular instruction format, we always need a native machine word to represent an instruction—with or without operand. The original, irregular instruction format, however, only needs a single byte for an instruction without operands, and an additional 2 bytes if that instruction has an operand. On a modern 64 bit architecture, where a native machine word corresponds to 8 bytes, this represents an overhead of 7 bytes per instruction in the first case and 5 bytes per instruction in the second case.

Benchmark	Encoded Instructions			
	Absolute		Percentage	
	Argument	No Argument	Argument	No Argument
ai-1	15,313	2,857	84.28%	15.72%
binarytrees-14	8,476	1,276	86.92%	13.08%
django-1	31,851	4,868	86.74%	13.26%
fannkuch-9	8,509	1,283	86.90%	13.10%
fasta-50000	8,625	1,306	86.85%	13.15%
mandelbrot-500	8,447	1,268	86.95%	13.05%
nbody-50000	8,698	1,417	85.99%	14.01%
spectralnorm-400	8,524	1,288	86.87%	13.13%
Total	98,443	15,563	86.35%	13.65%

Table 3.2: Distribution of instruction types using the irregular instruction set.

Table 3.2 shows that on average more than 85% of instructions have an operand, i.e., they require 3 bytes, whereas only about 15% of all instructions have no operand and therefore consume just one byte. Given these figures, we can easily calculate the additional space requirements of using our regular instruction

format. When 85% use 3 bytes and 15% use 1 byte, then one instruction uses 2.7 bytes on average ($0.85 \times 3bytes + 0.15 \times 1byte = 2.7bytes$).

On a 32 bit system, every instruction uses a native machine word, i.e., 4 bytes. Therefore, one instruction uses $4 - 2.7 = 1.3$ bytes more using a regular instruction set. This corresponds to an increase in space requirements by a factor of $\frac{4}{2.7} = 1.4815$, or about 50%. On a 64 bit system, all instructions use a native machine word, i.e., 8 bytes. Analogously to the 32 bit case, we need 5.3 bytes more per instruction, which corresponds to an increase in space requirements by a factor of 2.9630, or about 200%.

Thus, blindly switching to a regular instruction set for all instructions wastes valuable memory. Instead, our implementation uses a simple profiling technique for collecting cumulative calling and execution statistics. Whenever one of those numbers reach a configurable threshold, we choose to optimize the current activation. Thus, we can make informed decisions on when it is actually a good idea to trade space for time—without wasting valuable memory resources on infrequently executed parts.

3.2.1 Using Two Dispatch Routines

To support two distinct instruction formats, it is necessary to provide distinct instruction decoding for both formats. Because using a run-time switch to decide which decoding to use would offset our efficiency gains, we choose to use two separate dispatch routines, i.e., two routines that contain the same interpreter main loop, but with different instruction decoding. Note that doing this enables us to easily accommodate additional optimizations, such as using an extended instruction set: Whereas the original, system default, interpreter dispatch routine contains only the basic instruction set, the optimized dispatch routine can use an extended or even reduced instruction set. Therefore, the amount of cache misses caused by changing the instruction set remains the same when compared to the original interpreter, for infrequently executed parts.

So, we use two dispatch routines, one for supporting each instruction format. To decide which routine handles the current activation, we add an additional field to the internal code object data structure that represents Python code (`PyCodeObject`/`PyCode_Type`), named `co_inv_count`. We increment this field whenever we decide between the two routines. We choose to let the dispatch routine with the optimized instruction format be the default call target for all invocations. There, we check the `co_inv_count` field of the current code object whether it is above or below our configurable threshold. As long as the counter is below the threshold, we increase the counter and invoke the system default dispatch routine, i.e., the dispatch routine with the original, bytecode-based, irregular, instruction format. The rationale behind that choice is that the additional function call caused by using this setup is negligible for the infrequently called pieces of code it interprets.

```
...PyEval_EvalFrameEx...
if ( ! PyCode_HasReachedThreshold(PyFrame_GetCode(f)) ) {
    PyCode_IncCallCount(PyFrame_GetCode(f));
    return PyEval_EvalFrameEx_SysDefault(f, throwflag);
} // if
...
```

Once `co_inv_count` reaches the threshold, we allocate memory to hold our new instruction format, initialize this new memory area, and relocate the jump instructions. The instruction pointer related data types have to be adjusted as well, i.e., instead of being a pointer to `unsigned char`, our optimized interpreter uses the following data type definition for the instruction pointer:

```
typedef unsigned long bytecode_t;
```

Changing the de-referenced type of the instruction pointer requires to change the computation of the instruction offset, too.

3.2.2 Swapping the Current Execution

The previous section (Section 3.2.1) illustrates the standard profiling behavior that catches frequently called procedures. However, this type of profiling is unable to detect infrequently called functions that execute large amounts of bytecode instructions, such as top-level driver routines. Therefore, we complement the previous invocation based counter with a second profiling approach that approximates the total amount of bytecode instructions executed in the current interpreter main loop. When this total amount of bytecode instructions counter reaches a configurable threshold, we stop the current execution and swap it over to the optimized interpreter routine.

Approximating the Total Amount of Instructions Executed

A fairly simple way to count the number of instructions executed would be to increase a counter whenever we dispatch to the next instructions. This gives an accurate counter at the expense of causing considerable run-time overhead. Fortunately, however, the profiling technique does not necessarily require an accurate measurement but rather an approximate indicator enabling us to identify optimization potential. Even in the presence of a threaded code instruction dispatch optimization, the Python interpreter relies on some instructions doing a round-trip to the head of the dispatch loop for handling periodic tasks, e.g., releasing the global interpreter lock to enable the processing of other threads, or processing signal handlers. Every n instructions, the regular instruction dispatch is interrupted to take this round-trip. We use this functionality to roughly approximate the total number of bytecode instructions executed during the current activation.

The system default interpreter routine has an additional local variable that accumulates the approximation. Whenever our code runs, we add a fixed amount of estimated average instructions that have been executed so far. The following piece of code shows our implementation:

```
PyEval_EvalFrameEx_SysDefault:
...
unsigned long bytecodes= 0;
...
bytecodes+= _Py_CheckInterval;
if ((bytecodes >= OPT_PROFILE_NO_OF_INSTRS()) {
    /* swap execution to optimized dispatch routine */
} // if
```

Swapping the Current Execution

When we have reached the threshold of total bytecode instructions executed in the current activation, we swap its execution from the system default interpreter routine to its optimized counterpart. To swap the executions, we need to save the current execution state and ensure that the optimized interpreter routine can continue the execution where we stopped. The following two parts make up the current execution state:

- the instruction pointer,
- the stack pointer.

Whereas the latter can be used by both interpreter routines, it is precisely the change in the instruction format that forces us to relocate the former. This is, however, fairly simple: because the Python interpreter already supports co-routines in the form of generators, its stack frame object (`PyFrameObject`/`PyFrame_Type`) contains the offset of the last instruction executed (field identifier is `f_flasti`). Therefore, we just need to calculate the correct offset for the new instruction format, which consists of simply enumerating the current instruction position.

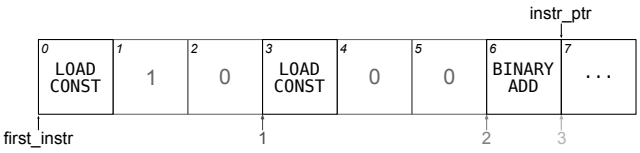


Figure 3.7: Relocating instruction pointer.

Figure 3.7 shows our implementation. Since the instruction pointer (`next_instr`) points to the following instruction (position: 7) and we will not need the pointer to the first instruction (`first_instr`) anymore, we can easily determine the new instruction offset by advancing the pointer while eliminating the need to decode the instruction operands. So, after decoding a new instruction—*not* its operand—, as Figure 3.7 shows below the bytecode sequence in sub-sequent steps 1, 2, and 3, we increment a counter for enumerating the number of instructions found (our implementation names this counter variable `lasti`). Thus, when we find that `first_instr` equals `next_instr`, i.e., both pointer have the same value (position 7 in Figure 3.7), we can relocate the instruction pointer to its new value by multiplying the value of `lasti` (which would be 2 in Figure 3.7) by the size of the native machine word in bytes (8 on a 64 bit architecture.) In our example of Figure 3.7, the relocated instruction pointer offset would be $2 * 8 = 16$.

The instruction pointer is, however, not the only place that requires relocation. The Python interpreter creates blocks (`PyTryBlock`) to handle block structures. The following listing shows its definition:

```
typedef struct {
    int b_type;         /* what kind of block this is */
    int b_handler;      /* where to jump to find handler */
    int b_level;        /* value stack level to pop to */
} PyTryBlock;
```

As we can see, it contains a field (`b_handler`) that holds the offset of the first instruction after that block (its identifier indicates the historic use for implementing exceptions, but they have been used for other mechanisms, too.) Since this offset value indicates an offset that includes the interleaved operand bytes, we have to relocate it. The following code shows how we are able to efficiently relocate all block handler offsets in just one pass:

```
#define BLOCK(elem)  f->f_blockstack[elem].b_handler
#define SKIP(flag)   (skipvector |= (2^flag));
#define SKIP_P(flag) ((skipvector & (2^flag)) != 0)
#define UPDATE(elem) (BLOCK(elem)= new_lasti * NEXT_ELEM_IN_BYTES());
#define COND_RELOCATE(BLOCK_NO) do { \
  if (!SKIP_P( BLOCK_NO )) BLOCK( BLOCK_NO )-= delta; \
  if (BLOCK( BLOCK_NO ) == 0) { \
    UPDATE( BLOCK_NO ); \
    SKIP( BLOCK_NO ); \
  }; \
 } while (0); \

int new_lasti= 0, i= 0, delta= 1, skipvector= 0, n= Py_SIZE(co->co_code);
while (i < n) {
    /* decode instr and advance instr-ptr */
    /* advances i, and adjust delta such that i':= i + delta */

    if (PyCode_GetNewInstrFormat(f->f_code) != NULL) {
        switch (f->f_iblock) {
            case 6: COND_RELOCATE(6); /* | */
            case 5: COND_RELOCATE(5); /* | */
            case 4: COND_RELOCATE(4); /* | */
            case 3: COND_RELOCATE(3); /* | */
            case 2: COND_RELOCATE(2); /* | */
            case 1: COND_RELOCATE(1); /* | fallthrough! */
            case 0: break;
        } // switch
    } // if
} // while

Py_LeaveRecursiveCall();
f->f_stacktop= stack_pointer;
PyCode_PromoteThreshold( f->f_code );
return PyEval_EvalFrameEx(f, throwflag);
```

Figure 3.8 illustrates the relocation algorithm of the previous listing. The left side shows the stack of `PyTryBlock`'s, having 3 `structs`.

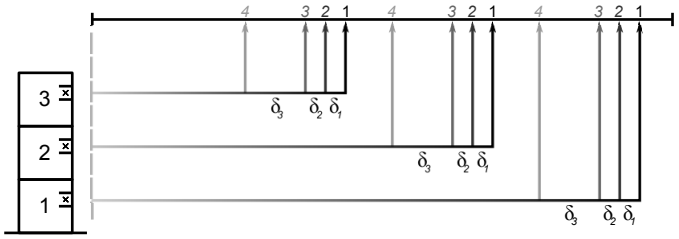


Figure 3.8: Relocating the stack of `PyTryBlock`s.

Within the `PyTryBlock`'s, the `x`'s denote the `b_handler` field containing the absolute offset of the first instruction *after* the block. In order to relocate, we

decrement the respective offsets until they finally become zero. In the listing, this is done in the while loop starting on line number 14. For every loop iteration, we determine the instruction size (either 1 or 3, depending on whether the instruction has operands) and store it in the variable `delta`. Next, we use the macro `COND_RELOCATE` to conditionally decrement the bytecode offset stored in `b_handler` by the instruction size stored in `delta`. This corresponds to the vertical arrows pointing to the decreasing offsets on the abstracted horizontal bytecode sequence line. Figure 3.8 indicates diminishing values by reducing their color from black to light-gray. The dashed vertically line on offset 0 corresponds to the value where we know exactly the instruction offset of the new instruction format. For this computation, we can reuse the `lasti` value from relocating the instruction pointer, i.e., the complete relocation procedure can be done in just one pass.

3.3 Inline Caching

Inline caching is a very important optimization that eliminates the overhead caused by dynamic typing. Originally described in the landmark work of Deutsch and Schiffman on the "Efficient Interpretation of the Smalltalk-80 System" [DS84], it has been refined in the early 90s by the introduction of polymorphic inline caches (PICS) by Hölzle and Ungar [HCU91, HU94], that allow the optimization of polymorphic call sites.

First, we will have a look on why inline caching actually works (Section 3.3.1). Next, we will show a purely-interpretative solution based on hash tables (Section 3.3.2). Following a discussion of an improved solution in a changed instruction format (Section 3.3.3), after which we present our most efficient solution that uses quickening to realize efficient purely-interpretative inline caching (Section 3.3.4). While we illustrate inline caching in the context of implementing arithmetic operation instructions, we finally present several additional applications of using inline caching in the Python interpreter (Section 3.3.5).

Please note that parts of this section have been published in prior work [Bru10a, Bru10c, Bru10b].

3.3.1 Dynamic Typing and its Locality

The interpreter of a dynamically typed programming language needs to resolve the ad-hoc polymorphism this feature imposes at run-time. This means, that the selection of the proper operation implementation of a given operation happens when the actual operand types are available—at the beginning of each operation.

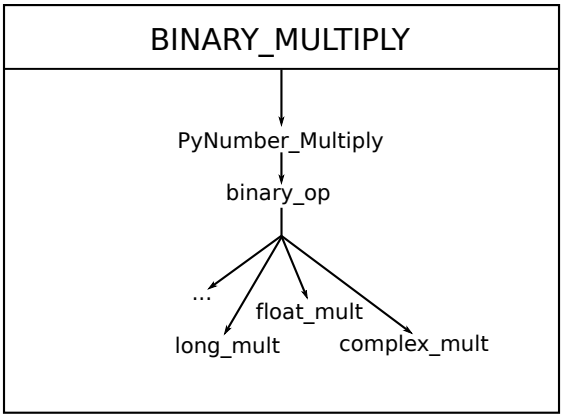


Figure 3.9: Resolving ad-hoc polymorphism.

Figure 3.9 illustrates an example of resolving the ad-hoc polymorphism imposed by dynamic typing using the `BINARY_ADD` instruction implementation. If we, for example, assume that the types of operand objects A and B are float objects (`PyFloatObject`/`PyFloat_Type`), the implementation of the `BINARY_ADD` instruction dictates the selection—and subsequent invocation—of the matching floating point addition function, i.e., the `float_add` function. Consequently, every execution of a type-dependent operation requires an equivalent resolving procedure.

It turns out, however, that once a specific occurrence of such an instruction has been executed, the operand types for that occurrence tend to remain constant for about 95% of all subsequent invocations of that occurrence. Deutsch and Schiffman call this the "observation of dynamic locality of type usage" [DS84]. Since the operand types remain constant with a very high likelihood, the repeated evaluation of the dynamic types for selecting the proper, type-dependent, operation implementation becomes *redundant* with the exact same likelihood. This constitutes the overhead usually associated with dynamically typed programming languages and their run-time systems.

Original Inline Caching Technique

Using this observation, Deutsch and Schiffman describe an optimization named "inline caching" that allows us to effectively eliminate the overhead in dynamic typing. The original description of inline caching by Deutsch and Schiffman uses a dynamic translation system, i.e., a just-in-time compiler in modern terminology. In Smalltalk-80, essentially every operation is a method call, or a message send in Smalltalk terminology—hence their interpreters contain a `send` instruction. When translating an interpreter's `send` instruction to native machine code, they emit a native machine call instruction invoking the *system default look-up routine*. Based on the operands to an occurrence of a `send` instruction, this routine selects the matching operation at run-time. Now, inline caching comes into play: After the interpreter identifies the actual address of the method to be invoked, it rewrites the original native machine call instruction to directly call the method, thus circumventing the redundant invocation of the system default look-up routine. To ensure correctness, we need to create guard statements that ensure our type assumption and will invoke the system default look-up routine whenever this assumption fails.

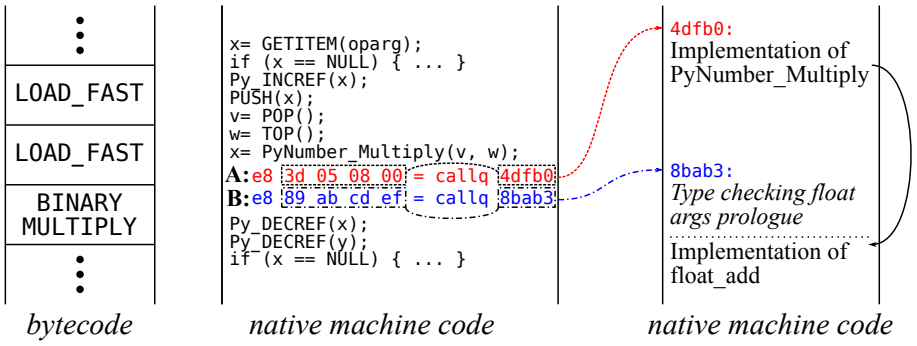


Figure 3.10: Inline Cache states.

Figure 3.10 illustrates the original inline caching technique, as described by Deutsch and Schiffman [DS84]. On the left, we see a sequence of bytecodes, i.e., virtual machine instructions—what Deutsch and Schiffman refer to as "*v-code*". A simple approach for compiling this sequence of bytecode instructions to native machine code would be to concatenate their respective operation implementations. In fact, this is what Figure 3.10 shows in the middle figure: the operation implementation of the second `LOAD_FAST` instruction and the

corresponding `BINARY_MULTIPLY` instruction are concatenated. This technique would eliminate the interpretative overhead due to the instruction dispatch. Now, whenever a dynamic compilation sub-system generates a function call, as is the case when we call the `PyNumber_Multiply` instruction, the call-site is said to be "*unlinked*". In Figure 3.10 this corresponds to the Intel x64 binary and assembly code displayed in red and indicated as position **A**. Invocation of the `PyNumber_Multiply` function would evaluate the actual operand types and for example end up invoking the `float_mult` function, i.e., the right-most arrow in Figure 3.10. Whenever a system default look-up routine choose an operation implementation based on the types of its actual operands, such as `PyNumber_Multiply` does, inline caching means that we "*link*" the corresponding call-site to the type-dependent function. In Figure 3.10 this corresponds to the blue binary and assembly code indicated as position **B**. Since the call-site in the native-machine code does not actually know which type the linked function expects, we have to link the call-site to a type-checking prologue that we prepend to the actual operation implementations. Figure 3.10 also shows what inline means: we inline the target address into the native machine instruction sequence.

PICS and JITs

Following this very important contribution by Deutsch and Schiffman in 1984 [DS84], Hölzle, Chambers and Ungar introduced an important derivation of this technique: polymorphic inline caches, or PICS for short, in 1991 [HCU91]. While this work is highly relevant and has been put to successful use in many systems, such as the Java virtual machine's just-in-time compiler, it has a major drawback: the necessity of having a dynamic compilation sub-system.

While the implementation of a just-in-time compiler is itself a non-trivial, time consuming task, its detrimental effects on future maintenance and portability of a programming language implementation impose considerable resource requirements for any project. History shows that a successful implementation of a just-in-time compiler is a multi-year, multi-person effort that is usually reserved for corporations. If a project has enough visibility in the open source world, these costs can be offset by volunteering implementers. Recent research [YWF09, BCFR09] addresses these issues by investigating techniques for leveraging existing virtual machine technology by implementing an interpreter on top of an existing virtual machine that already offers a just-in-time compiler—this is somewhat similar to the frontend/backend abstraction in compilers. While these results present promising first steps in a new direction, we think that it is certainly worthwhile to investigate purely interpretative optimization techniques complementing these approaches.

3.3.2 Look-up Caches

One very popular, purely interpretative, approach to mimic the original inline caching approach is to use hash-tables as look-up caches. Usually, globally accessible hash-tables with a fixed size of entries contain method addresses for a given pair of class type and method name, or `selector` in Smalltalk terminology. If the look-up is successful, this technique requires an indirect branch to call the target method address. Otherwise, the technique requires us to call the system default look-up routine and place its result in the hash-table. This is a

worthwhile optimization under the premise that the indirect branch combined with the look-up costs is less expensive than the actual invocation of the system default look-up routine.

While this technique is advocated by the Smalltalk 80 implementers guide [GR83] it has been used (often with changes) in other systems, for example the Portable Common Lisp implementation [KR90], as well.

A good deal of research has gone into investigating this technique, the following articles from "Smalltalk-80: Bits of History, Words of Advice" contain valuable information [Kra84]:

- Allen Wirfs-Brock, "Design Decisions for Smalltalk-80 Implementors", Chapter 4 [WB82]: lists a couple of favorable characteristics for use of these look-up caches.
- Thomas J. Conroy and Eduardo Pelegri-Llopart, "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations", Chapter 13 [CPL82]: Analyzes the accuracy and efficiency for several hashing methods to be used when looking up information in hash-tables.

3.3.3 Interleaving Inline Cache Pointers

Another, more efficient, purely interpretative way of using dynamic locality of type usage for optimization is to interleave the interpreter instruction sequence with native machine words. Using these additional words, we can save addresses to functions for any instruction occurrence.

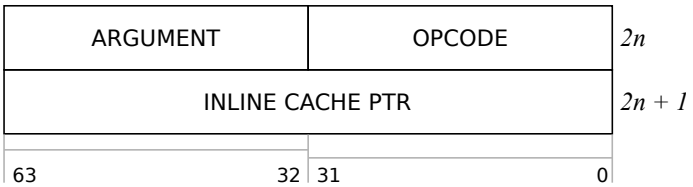
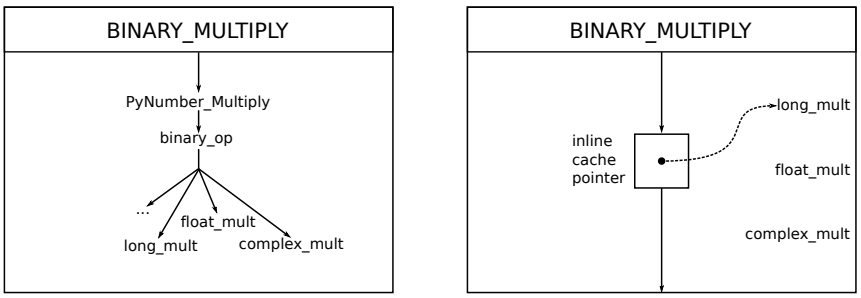


Figure 3.11: Interleaving inline cache pointers.

Figure 3.11 illustrates how this change affects our instruction format. Interleaving the words requires us to change the instruction decoding: Since instructions are located at even positions ($2n$), and inline cache pointers at odd offsets ($2n + 1$), the instruction decoding must step over these inline pointers to point to the successor instruction. We can use the relocation procedure mentioned in Section 3.1.1 to adjust the jump target offsets.

Figure 3.12 shows how we can use this technique to achieve purely interpretative inline caching. We see in Figure 3.12 that the inline cache pointer allows to replace the call to the system default look-up routine (`PyNumber_Multiply`) with an indirect branch to the target address referenced by the inline cache pointer. In consequence, we need to ensure that when we create the new instruction format, the inline cache pointers are properly initialized, for example that every `BINARY_MULTIPLY` occurrence gets its inline cache pointer initialized to `PyNumber_Multiply`.

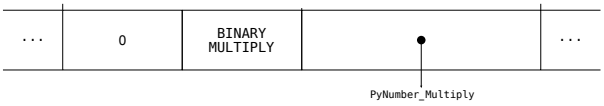
To incorporate the type feedback at run-time, we need to update the inline cache pointer of the current instruction accordingly. As we can see in the control



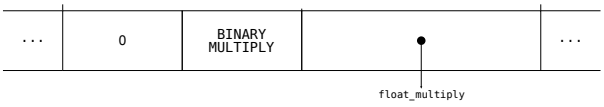
(a) Ad-hoc polymorphism in Python 3.1. (b) Basic inline caching `long_mult`.

Figure 3.12: Illustration of inline caching using interleaved inline cache pointers.

flow of the `BINARY_MULTIPLY` instruction (cf. Figure 3.12a), the `binop` function selects the operation implementation based on actual type information, i.e., converges from type-generic to type-dependent operation implementation. Therefore, we instrument the `binop` function to update the inline cache pointer whenever it has successfully determined the type-dependent operation implementation, given some arbitrary but fixed operand types (cf. Figure 3.12b).



(a) Inline cache pointer initialized to invoke system default lookup routine.



(b) Updated inline cache pointer.

Figure 3.13: Purely interpretative inline caching using interleaved pointers.

Figure 3.13 demonstrates how the update affects subsequent execution of a specific occurrence of an instruction. While the first execution of that occurrence invokes the system default look-up routine (cf. Figure 3.13a), updating the corresponding inline cache pointer eliminates the redundant evaluation of dynamic types (cf. Figure 3.13b).

Dealing with Cache Misses

The previous section contains details of how the inline caching approach using interleaved pointers works. However, as we have mentioned earlier (cf. Section 3.3.1), the cache is only valid about 95% of the time, i.e., we still need to explain what happens in the remaining 5% of cache misses. The inline cache references a type-dependent function where the actual operand types do not match the previously valid assumption anymore. One way to prevent this from

happening would be to add another set of words to the instruction format and encode the expected types there. For example, for a binary instruction, we would need to add two words to the instruction format to encode the operand types for both operands. Unfortunately, this change has two undesirable properties:

- increased memory consumption,
- implies an irregular instruction format.

While the first may seem to be bigger issue among the two, the latter one is actually more problematic. We have already described the downside of irregular instruction decoding in Section 3.1.1, but there is another major issue: a regular instruction format enables us to easily find the inline cache pointers for each instruction, such that the actual updating of its contents becomes trivial. In an irregular instruction format, *efficient* updating the inline cache pointer becomes difficult.

There is, however, a very simple way of dealing with cache misses eliminating both problems. While the interpreter operation implementation, i.e., the caller, has no information of the types expected by the type-dependent function, this type-dependent function, i.e., the callee, is perfectly aware of the types it expects. Therefore, we can prefix this type-dependent operation implementation function with a type check to identify and handle cache misses. The following listing shows our implementation of such a prefix inline cache miss check:

```
static PyObject *
long_add(PyLongObject *a, PyLongObject *b) {
  if (!PyLong_Check(a) || !PyLong_Check(b)) {
    return inca_binary_add_sysdefault(a, b);
  } // if
    ...
}
```

Invoking the type-generic operation implementation function will re-evaluate the operand types and select another operation implementation. Since this path passes our instrumented operation selection function, this takes care of updating the inline cache pointer for this operation, too.

Comparison with Look-up Caches

This technique eliminates the overhead implied by using a hash-table data structure, i.e., we need no hashing, no look-up procedure, etc. Furthermore, we are improving data locality by eliminating the access of the memory location of the hash-table. While it is possible that the access of the hash-table memory location may cause a page-fault, it is not possible using our technique. It is certainly reasonable to assume that the hash-table, or—depending on its size—at least some of its contents, will be present in hardware caches. Since our technique eliminates the hash-table, we free up these resources, such that they can support the actual interpretation.

The major issue with using this technique is its additional impact on memory consumption. It turns out, however, that we can compensate for these additional memory requirements by combining this interleaved representation with our profiling technique (cf. Section 3.2).

Historical Perspective

Urs Hölzle's PhD thesis [Höl94] of 1994 contains some interesting material regarding the importance of several optimizations for the performance of a SELF interpreter in Chapter 4.5 on page 31:

> *Without inline caching, sends are expensive.* A straightforward interpreter could not use inline caching and would have to use a lookup cache instead.

However, the last paragraph in the same chapter on page 32 reads as follows:

> Alternatively, the interpreter could just add inline caching to the straightforward interpreter by adding one pointer per send byte code to cache the method last invoked by that send; each method would also cache the last receiver type. Similar organizations have been used for Smalltalk interpreters [44 *(ed: [DS84])*]. Since most byte codes are sends, this approach would require roughly one word per byte code.

With only minor changes, this describes the approach presented herein, too. The first difference is that we propose to add an inline cache pointer to every interpreter instruction, whereas Hölzle describes this addition only for `send` instructions. This, however, is not a major difference in ideas, but rather comes from the difference in interpreters: SELF requires an even higher abstraction-level virtual machine than Python does, since—in the end—everything is resolved using `send` instructions, whereas the Python interpreter contains numerous instructions that do not rely on a method call[1]. Therefore, it would be quite unnecessary to add the inline cache pointer to every instruction, except for having a regular instruction set—which could very well be worthwhile, because most instructions are in fact `send` instructions anyways. The second difference is that we propose the combination with profiling to limit the somewhat excessive memory requirements implied by adding the inline cache pointers—there is no hint for a similar architecture in these paragraphs.

Hölzle mentions that a similar approach has been used in Smalltalk systems before and refers to the "Efficient Implementation of the Smalltalk-80 System" of Deutsch and Schiffman. However, the author could not find any such references in that work. In addition to that, this technique has not been implemented by Hölzle, such that we cannot compare our results.

In private communication, Urs Hölzle [Höl09] as well as Gilad Bracha [Bra10] pointed out that a similar architecture was used by the Strongtalk interpreter. After the hint by Urs Hölzle and its followup analysis, the author incorrectly believed that a similar inline caching architecture was used only in the case of just-in-time compilation, i.e., during native code execution. The comment by Gilad Bracha led to another analysis that indicated that the Strongtalk interpreter did in fact use the same kind of inline caching for `send` instructions. The source code in question (see below) identifies Robert Griesemer is the author of that particular piece of code [Gri96]—Gilad Bracha independently confirmed this.

[1] Note that because Python supports additional ad-hoc polymorphism via operator overloading, thus most interpreter instructions *can* end up calling a method.

```
//------------------------------------------------------------------------------------------
// Inline cache structure for non-polymorphic sends
//
// [send byte code ]      1 byte
// [no. of args    ]      1 byte, only if arg_spec == recv_n_args
// alignment .....        0..3 bytes
// [selector/method]      1 word
// [0/class        ]      1 word
// next instr ...... <--- esi, after advance
//
// normal_send generates the code for normal sends that can
// deal with either methodOop or nmethod entries, or both.
//
// Note: As of now (7/30/96) the method sweeper is running asynchronously and might modify
//       (cleanup) the inline cache while a send is in progress. Thus, the inline cache might
//       have changed in the meantime which may cause problems. Right now we try to minimize
//       the chance for this to happen by loading the cached method as soon as possible, thereby
//       reducing the time frame for the sweeper (gri).
```

In addition to the normal inline caching, the Strongtalk interpreter uses purely interpretative inline caching, polymorphic inline caches, as well as megamorphic inline caches, too. Contrary to our approach, it does not seem that the Strongtalk interpreter combines this architecture with profiling—rather the profiling is used to trigger the just-in-time compilation of frequently called methods. Unfortunately, there is no publication highlighting the Strongtalk interpreter as the first implementation of purely interpretative inline caching using interleaved pointers. Therefore, the author independently re-discovered this idea about 13 years later and subsequently published this result [Bru10a, Bru10b, Bru10c]—the prior work being unbeknownst to the author at that point.

3.3.4 Quickening

Quickening is a very important run-time optimization technique for interpreters. It is based on using information obtained at run-time and rewriting generic instructions to optimized derivatives of these generic instruction implementations. We will first describe the original quickening optimization technique as it is used in a Java virtual machine. Next, we introduce our novel approach to use quickening for inline caching. Finally, we will present a brief discussion of the pros and cons of using this optimization technique.

Quickening in the Java Virtual Machine

The Java virtual machine contains several instructions, such as the `getfield` and `putfield` instructions, that reference data from the constant pool. The Java virtual machine specification [LY96], describes how the instruction implementation of those instructions dynamically resolve their respective symbolic constant pool entries. The result of this resolving procedure is invariant with respect to recurring execution. Therefore, it is only necessary to actually resolve constant pool entries once and cache their results for subsequent executions. Now, optimized derivatives of these instructions omit expensive resolving steps and directly access the cached result data. So, the Java compiler generates the generic instructions as defined by the JVM specification. After their first execution, they rewrite themselves to the optimized derivatives, such that these expensive resolving procedure needs to be executed only once per occurrence. This quickening optimization is internal to the interpreter, i.e., it is not necessary

for the compiler to know about the additional instruction derivatives. Summing up, the quickening interpreter optimization consists of the following parts:

- Creating optimized instruction derivatives.
- Rewriting from generic to optimized derivatives.

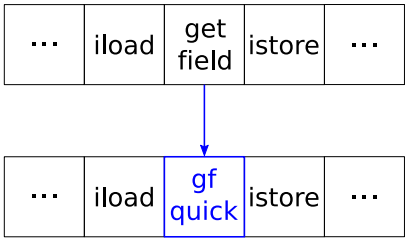
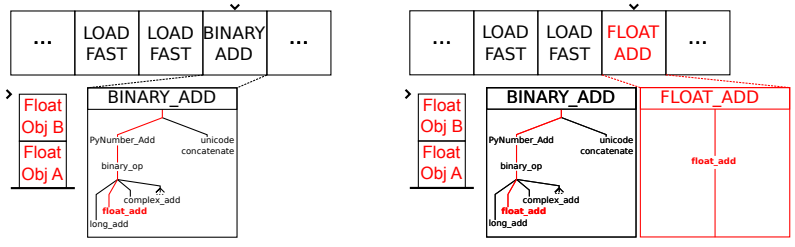


Figure 3.14: Quickening in the Java virtual machine.

Figure 3.14 shows how this replacement occurs. On the left side, we see how the generic instruction resolves its reference to the constant pool. The operand value of the instruction refers to a symbolic constant pool entry. The interpreter caches the result of resolving this symbolic entry, and rewrites the operand value to identify the cached data object. Hence, the derivative relationship between the generic instruction and its derivative is defined via the operand value.

Inline Caching meets Quickening

The previous sections describe various important optimization techniques, viz. inline caching and quickening. Now, we show how a combination of both techniques results in a more efficient optimization technique. As we have mentioned above, the JVM uses quickening to provide optimized derivatives with respect to an operand value. If we, however, provide optimized derivatives with respect to the result of resolving the ad-hoc polymorphism imposed by dynamic typing, quickening allows for efficient inline caching in an interpreter.



(a) Resolving ad-hoc polymorphism. (b) Inline caching using quickening.

Figure 3.15: Purely interpretative inline caching using quickening.

Figure 3.15 illustrates this approach: while executing the generic `BINARY_ADD` instruction, the `binary_op` function selects the proper operation implementation for the actual operand types. Now, we quicken the generic `BINARY_ADD`

instruction to an optimized derivative that hard-wires the exact same operation implementation function—in our case, the `INCA_FLOAT_ADD` instruction that directly calls the `float_add` function. Thus, every subsequent execution of this `INCA_FLOAT_ADD` function will by-pass the redundant type resolving mechanism. This pretty much resembles the original approach of inline caching in the Smalltalk-80 system of Deutsch and Schiffman, with the notable exception that we lifted the rewriting to the interpreter level instead of using a dynamic translation sub-system.

In the interleaved pointer technique of Section 3.3.3, we are detecting cache misses by pre-pending our type assumption checks to the actual operation implementation, i.e., pre-pending `PyFloat_Type` check before executing the rest of the body of the `float_add` function. This was necessary, because the `BINARY_ADD` instruction had no knowledge of our type assumptions—it was lacking this context. By using this quickening based inline caching approach, however, every optimized derivative has this context, and therefore we can generate these cache-miss guard statements directly in the operation implementation, without the need to instrument numerous library functions.

```
TARGET(INCA_FLOAT_ADD)
    /* check for inline cache miss */
    if (!(PyFloat_Check(TOP()) && PyFloat_Check(SECOND())))
        /* call the default implementation without instr. decoding */
        goto TARGET_BINARY_ADD_SKIP_DECODE;

    w= POP();
    v= TOP();
    x= PyFloat_Type.tp_as_number->nb_add( v, w );

    Py_DECREF( w );
    Py_DECREF( v );

    SET_TOP(x);

    if (x != NULL) DISPATCH();
    break;
```

Discussion

Lets discuss the advantages and disadvantages of both inline caching techniques presented in the previous sections. The most important difference between both techniques is that the quickening based approach eliminates the necessity of interleaving native machine words with the actual instructions. Thus, the quickening based inline caching requires only half as much memory as the interleaving pointer technique does. However, since we extend the instruction set with the optimized derivatives in the quickening based optimization technique, the maximum representable 255 instructions in a bytecode format can pose a major problem. Therefore, a change of the interpreter's instruction format is recommended when using either one of the inline caching techniques.

Along with the reduction of memory requirements, the quickening based technique eliminates the indirect branch instruction by hard-wiring the operation implementation function. While this elimination is beneficial on its own, it enables the compiler to inline the function body of that function into the operation implementation of the interpreter instruction, too. This inlining will

provide further performance improvements, since it considerably reduces the function call overhead of frequently executed interpreter instructions.

Unfortunately—as is the case with many optimizations—, the quickening based inline caching technique has disadvantages as well. First of all, extending the interpreter instruction set with the multiple instruction derivatives presents considerable "noise" for any programmers dealing with the interpreter. Our solution to this problem is to use a code generator to generate the multiple derivatives in a pre-compile step (cf. Section 3.7). Thus, the actual interpreter implementation remains largely untouched and reduces the "noise" to a bare minimum. Another downside of using quickening based inline caching is that its net-performance depends on instruction cache size of the underlying CPU. If we, for example, assume a small-scale CPU with only a small instruction cache, it is possible that the larger interpreter instruction set causes more instruction cache misses. The implied cache miss penalties could overcompensate for the efficiency gains of using quickening based inline caching. So, while the quickening based inline caching technique achieves the biggest speedups on modern desktop and server CPUs, it is possible that the interleaving pointer approach outperforms the quickening based optimization on smaller CPUs. Interestingly, this quickening based inline caching techniques performs significantly better on all of our evaluation systems.

Finally, it is important to note that while the original quickening in the JVM happens only once per occurrence, the inline cache based quickening technique requires re-quickening after every inline cache miss.

3.3.5 Inline Caching Applications

During the previous sections, we illustrated our inline caching techniques using the implementation of arithmetic operation instructions of the interpreter. The use of this technique is not, however, restricted to this type of instructions. The following sections provide additional details on using inline caching for other instruction types.

Optimizing Call Instructions

Like many other modern programming languages, Python promotes the use of (imperative) functions as its primary abstraction mechanism. Note that Python is a multi-paradigm programming languages, which means that functions can be methods as well. Therefore, the call instruction is among the most frequently used interpreter instructions. Unfortunately, the `CALL_FUNCTION` instruction is very *abstract*, similar to Urs Hölzle's assessment of the `send` instruction in the SELF interpreter ([Höl94], page 31):

> *The byte code encoding is very abstract.* As discussed above, there is only one `send` byte code, and even local variable accesses are expressed with this byte code. For efficient interpretation, it is probably necessary to redesign the byte code format to separate the "trivial" sends from the "real" message sends, or to cache an intermediate representation as discussed below.

Though Python does not use the `CALL_FUNCTION` instruction to access local variables, it is still too abstract for efficient interpretation, since it supports the following possible call targets:

- Python functions,
- Python methods,
- C functions.

Thus, every execution of a `CALL_FUNCTION` instruction requires the interpreter to first determine the call target, before it can actually call it. However, dynamic locality of type usage corresponds in the `CALL_FUNCTION` case to very infrequent change of call targets. In fact, to change the call target of a specific `CALL_FUNCTION` occurrence one would have to explicitly code it that way.

Besides supporting several call targets at once, Python allows for special cases when invoking C functions with only no arguments or only one argument. The reason for this is that in general, Python represents function arguments by re-using its own tuple type (`PyTupleObject`/`PyTuple_Type`). Passing `NULL` instead of a reference to an argument tuple is a shorthand for calling a C function that expects no arguments. Similarly, passing the Python object directly without the wrapping tuple object represents calling a C function with only one argument.

Finally, Python supports several additional peculiarities for passing arguments to functions: a) named arguments, b) variable length argument lists, and c) a combination of a) and b). Figure 5.2 in the evaluation chapter (cf. Section 5.2.3) contains the result of a simple quantitative analysis of dynamic instruction, operand value and call target frequencies. Based on these information, we provide optimized derivatives that replace the generic `CALL_FUNCTION` instruction for each of the combinations in Table 3.3.

Target	Number of Arguments			
	Zero	One	Two	Three
C std. args	x	x		
C variable args	x	x	x	x
Python function	x	x	x	
Python method	x	x	x	

Table 3.3: Optimized `CALL_FUNCTION` derivative instructions.

Optimizing Iteration Instructions

The Python interpreter contains a dedicated instruction for iteration, `FOR_ITER`. This instruction expects an iterator object as the top-of-stack element, and it will call a function on that iterator object resulting in either, the next value object for another iteration, or a `StopIteration` exception, in which case it will remove the iterator object from the operand stack. The following listing shows the implementation of the `FOR_ITER` instruction.

```
TARGET(FOR_ITER)
    /* before: [iter]; after: [iter, iter()] *or* [] */
    v = TOP();
    x = (*v->ob_type->tp_iternext)(v);
    if (x != NULL) {
        PUSH(x);
        DISPATCH();
    }
    ...
```

Dynamic locality of type usage for the iterator instruction means that the actual type of the iterator object used for a given occurrence of a `FOR_ITER` instruction is very likely to remain constant. Actually, in this case, we see that the operand stack layout ensures that as long as the iterator sequence continues, not only the type will remain the same, but the actual iterator object will remain the same. Therefore, the repeated resolving of the indirections identifying the function that takes this iterator data object reference and computes its successor data object reference is completely redundant. Using the quickening based inline caching approach allows us to create several optimized instruction derivatives that hard-wire the iterator computation function. This allows the compiler to perform inlining and removes the indirections as well as the indirect branch. The following figure illustrates the optimization that is possible in that case:

```
TARGET(FOR_ITER_RANGEITER)
    v= TOP();

    /* check for inline cache miss */
    if (v != NULL &&
        (unsigned long)v->ob_type != (unsigned long)&PyRangeIter_Type)
        goto TARGET_FOR_ITER_SKIP_DECODE;

    /* direct call to inline cached function */
    x= PyRangeIter_Type.tp_iternext( v );

    if (x != NULL) {
        ...
    }
    ...
```

Analogously to the case with inline caching the arithmetic operator instructions, we can easily detect a cache miss by the context present in each instruction derivative. In such a case, we literally `goto` the generic `FOR_ITER` instruction, which determines the new iterator object type and re-quickens itself to another optimized instruction derivative.

Optimizing Comparison Instructions

Like most other instructions of the Python interpreter, the comparison instruction relies on ad-hoc polymorphism as well. Most often, the `COMPARE_OP` instruction will invoke a type-generic comparison routine, `do_richcompare`, which uses the type information of both operands to select the actual type-dependent comparison routine. Again, the dynamic locality of type usage allows us to use inline caching for optimization here. By providing optimized derivatives with hard-wired calls to the comparison routines, we are able to eliminate the repeated, yet redundant, resolving of the ad-hoc polymorphism.

```
TARGET(INCA_CMP_FLOAT)
    w= POP();
    v= TOP();

    if ((v->ob_type != w->ob_type)
        || (((size_t)v->ob_type) != (size_t)&PyFloat_Type))
        goto COMPARE_OP_MISS;

    x= PyFloat_Type.tp_richcompare(v, w, oparg);

    Py_DECREF( w );
    Py_DECREF( v );
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
```

3.4 Reference Counting

In the previous section (Section 3.3) we describe in-depth how to efficiently implement purely interpretative inline caching techniques to eliminate the overhead in dynamic typing. Unfortunately, however, this is not the only deficiency of the Python interpreter: as we have described in Chapter 2, there are two remaining bottlenecks in the operation implementation, viz. reference counting and (un-)boxing of data objects. In this section we present a novel optimization technique to considerably optimize reference counting. This technique presents an alternative approach to the landmark work of Deutsch and Bobrow in 1976 [DB76], and maintains some of the good parts of immediate reference counting, while eliminating the downside of deferred reference counting.

Immediate Reference Counting

Introduced in 1960 by Collins [Col60], reference counting provided a solution to a problem with the mark and sweep garbage collection technique described by McCarthy [McC62], viz. mutator pauses consuming too much time.

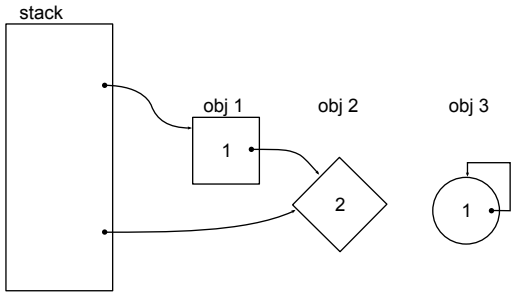


Figure 3.16: Illustration of immediate reference counting [Ung86].

The basic reference counting invariant is that an object maintains a count of objects referencing it at all times. Figure 3.16 illustrates how all objects correctly enumerate the number of objects referencing it. Reference counting is a very interesting automatic memory management technique that has the following advantages:

- reclaims data immediately when it becomes garbage,
- storage maintenance is roughly equivalent to the overall computational effort,

On the other hand, the following disadvantages explain why the technique has become increasingly unpopular since its original description:

- cannot reclaim circular data references (cf. circular reference of obj 3 in Figure 3.16),
- accommodation of the reference count datum,
- overhead of recursive freeing on container objects,
- large amounts of reference count operations due to local operand stack modifications.

Deferred Reference Counting

As early as 1976, Deutsch and Bobrow present an optimization technique to solve one of the biggest problems with the original reference counting approach, viz. the huge amounts of reference count operations caused by local operand stack modifications [DB76].

Deutsch and Bobrow observe that it is actually not necessary to continuously adjust the reference count of all data objects because of these local operand stack operations. In fact, they present a way to determine correct reference counts by using a sophisticated scheme of tables that contain references to several data objects. In consequence, they introduce a dedicated deferral phase, where they process all pending reference counts and free objects that become garbage during that process. Thus, all reference count operations are deferred to a dedicated phase, similar to a mutator pause that other garbage collection techniques require.

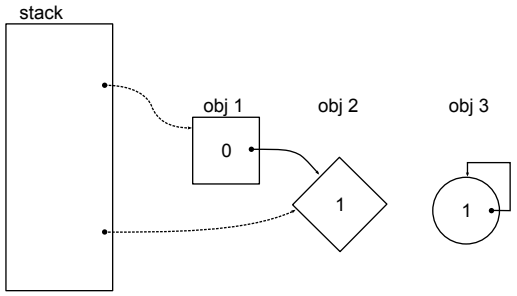


Figure 3.17: Illustration of deferred reference counting [Ung86].

3.4.1 Interpreter Operations Causing Reference Count Operations

Scott Baden's measurements of 1982 [Bad82] indicate that the deferred reference counting approach of Deutsch and Bobrow eliminates about 90% of reference count operations. This demonstrates that most of the reference count operations are due to modifications of the local operand stack. In order to present our alternate solution to this problem, we first need to clarify why local stack operations cause so many reference count operations.

Example

Figure 3.18 shows a simple sequence of three bytecode instructions. On the left side, we see the operand stack with two operand objects, `A` and `B` pushed onto the stack by the preceding `LOAD_FAST` instructions. Now the operand stack has references to these objects, which is why the implementation of `LOAD_FAST` increments the reference counts of these objects. The implementation of `BINARY_ADD`, however, shows that after the operands are fetched from the operand stack (they got assigned to the local variables `v` and `w`) and added using the `PyNumber_Add` function, their reference counts are immediately decremented. Consequently, in this sequence of three instructions, four reference count operations (one

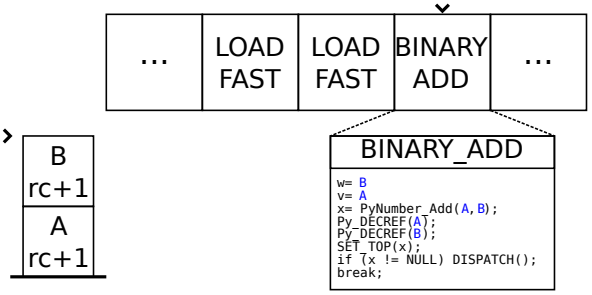


Figure 3.18: Redundant reference count operations.

increment in each `LOAD_FAST` operation implementation plus two decrements in the `BINARY_ADD` operation implementation) are *redundant*—their presence is due to the conservative operation implementation.

Operand Stack Operations

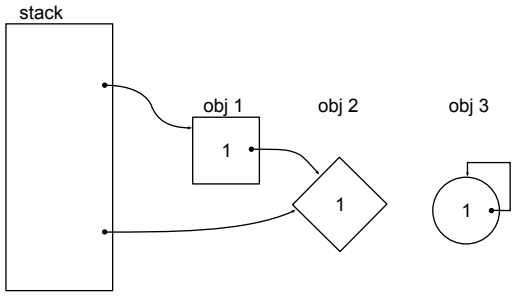


Figure 3.19: Illustration of redundant reference count operations.

Analogously to the visualization of immediate and deferred reference counting (cf. Figures 3.16, and 3.17 respectively), Figure 3.19 displays how the interpreter can eliminate redundant reference count operations: The second object has a reference count of just one, while actually two references point to the object. The second reference, however, is a redundant reference, i.e., it does not actually affect the reference count of the object (obj 2). A load instruction would increment and a subsequent operation instruction decrement it, therefore, only the reference pointed to by the first object (obj 1) counts.

3.4.2 Simple Abstract Interpreter

Following the description of Java bytecode verification [Ler03] using a simple abstract interpreter over the operand types, we realized that we can use the same approach to identify sequences of redundant reference count operations: by using a simple abstract interpreter over the amount of reference count operations present in an operation implementation.

Implementation

The following listing contains our basic data type definitions:

```
typedef struct {
  signed char imp; /* implicit rc ops */
  signed char exp; /* explicit rc ops */
} tuple_t;

typedef struct {
  bytecode_t *instrPtr;
  tuple_t effect;
} effect_stack_elem_t;
```

We use the first data type `tuple_t` to model the reference count effect of one instruction, i.e., the number of reference count operations that occur in one operation implementation. Positive values indicate the number of increment reference count operations, while negative values enumerate the number of decrement reference count operations present in the operation implementation. Furthermore, we need to distinguish between explicit and implicit reference count operations—the following section (cf. Section 3.4.3) explains why this is necessary.

In addition to this tuple, our simple abstract interpreter stores a reference to the instruction that generated this reference count effect. Thus, whenever we find a sequence of redundant reference count operations, we can easily quicken instructions to their optimized derivatives using this sequence of pointers.

Next, we declare the local variables our simple abstract interpreter uses:

```
bytecode_t *instr_ptr= codeobject->co_opt_code;
bytecode_t *cur= instr_ptr;
effect_stack_elem_t *stack_ptr= stack;
tuple_t effect;
opcode_t opcode;
int i= 0, n= 0, size= Py_SIZE(codeobject->co_code);
```

`Instr_ptr` always points to the next instruction, and because the Python interpreter has an irregular instruction format, we need another pointer, `cur`, to keep a reference to the current instruction. `Stack_ptr` points to a globally pre-allocated stack of the previously explained custom data-type `effect_stack_elem_t`. We use the `effect` variable to determine the reference count effect the current operation has. When decoding an instruction using the `instr_ptr`, we determine the `opcode` of an instruction as a by-product. To iterate over all instructions of a Python function/method, we use the variables `i`, and `size` respectively. We store the arity of an instruction in the variable `n`.

```
1  while (i < size) {
2    cur= instr_ptr;
3    opcode= decode_instr();
4    i++;
5
6    if (basic_block_border(opcode))
7      clear_stack();
8
9    if (rotate_instr(opcode)) {
10     rotate_stack_elems(stack_ptr, opcode);
11     continue; /* skip to the next iteration */
12   }
13
14   if (refcount_effect(opcode, &effect)) {
15     if (effect.exp < 0) {
16       n= -effect.exp;
17       stack_ptr-= n;
```

```
18
19        if (opcode == CALL_FUNCTION) {
20          ...implementation omitted...
21        }
22        else {
23          if (n <= 2 && is_markable(opcode)) {
24            if (n == 2)
25              quicken_binary_op(&cur, &stack_ptr);
26            else if (n == 1)
27              quicken_unary_op(&cur, &stack_ptr);
28          }
29        }
30      }
31      push_opnds(&cur, &stack_ptr, &effect);
32    }
33  }
```

Example

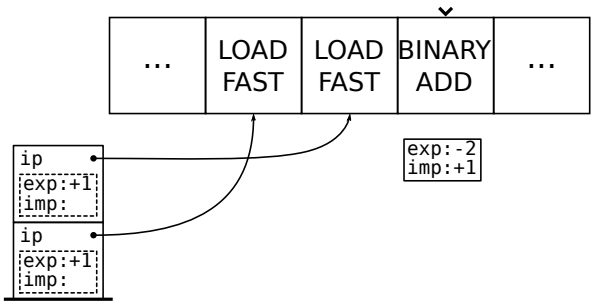


Figure 3.20: Finding redundant reference count operations using an abstract interpreter.

Figure 3.20 shows an illustrative example of an instruction sequence with redundant reference count operations. We see how the simple abstract interpreter we have just described maintains a stack of `effect_stack_elem_t` elements. The `ip` member of the `effect_stack_elem_t struct` points to the instructions that produced the stack element and amount of reference count operations information (line 31 of the listing). When we determine that the `BINARY_ADD` instruction has a stack effect of explicitly minimizing the reference counts of its two operands (line 14 of the listing), we have successfully identified a redundant sequence of reference count operations.

Figure 3.21 illustrates the elimination of redundant reference count operations by quickening the member instructions of the sequence to their corresponding optimized derivatives. First, we decrement the stack pointer and assign the operation arity to the variable ($n := 2$, lines 16 and 17.) We find that we are quickening a binary operation (lines 23 and 24), and call the corresponding function with references to the current instruction and the stack (line 25). These references are necessary to quicken the instructions to their optimized derivatives—indicated by the new instruction identifiers (postfix `NORC`) in a red font of Figure 3.21.

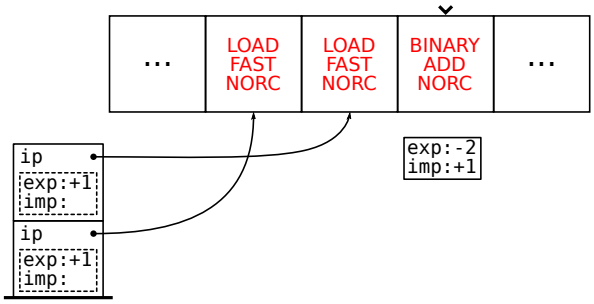


Figure 3.21: Elimination of redundant reference count operations by quickening.

3.4.3 Quickening

The previous section explains how our simple abstract interpreter identifies sequences of redundant reference count operations. In Figure 3.21 we have already illustrated how we use quickening to actually eliminate redundant reference count operations. Therefore, we have to pre-generate optimized derivatives *without* their corresponding reference count operations, such that we can use them for quickening. For unary instructions, we need just one derivative omitting just the one reference count operation that is present. Unfortunately, however, this is not the general case. For example, whenever we want to eliminate reference count operations for binary instructions, we have to consider four cases instead of just two:

- Perform all four reference count operations—this corresponds to the conservative default implementation.
- Eliminate all four reference count operations—this represents the ideal, and actually most frequent, case.
- Eliminate just two reference count operations—this can be the case for either:
  - the first operand, or
  - the second operand.

Therefore, we have to carefully consider dynamic bytecode frequency measurements to make informed decisions for which binary instructions it pays off to have all three necessary derivatives. Making informed decisions is even more important in the case of ternary operations and function calls, which have variable arity. However, in the last case, we have observed that the following cases occur most often:

- Perform all n reference count operations for all n operands,
- Eliminate all n reference count operations,
- Eliminate $n - 1$ reference count operations, i.e., all but the top-of-stack operand.

Thus, using just two extra `CALL_FUNCTION` derivative instructions, we can cover the most frequently executed function call redundant reference count operations. We perform the necessary calculation in line 20 of the simple abstract interpreter listing, but omitted it from the presentation because of space issues.

To select the proper derivative among the variations, we have to distinguish between explicit and implicit reference count operations. If we take a closer look at the `BINARY_ADD` implementation of Figure 3.18, we see that the reference count of the result of the addition (held in variable `x`) is never incremented. Therefore, any subsequent reference count operation cannot be redundant, i.e., a corresponding decrement reference count operation is necessary. We call such a reference count operation *implicit* reference count operation, because it is not visible in the operation implementation of the instruction, it is implicitly done inside the operation implementation function (`PyNumber_Add` in Figure 3.18).

3.5 Partial Stack Frame Caching

Some parts of this section have previously been published in the author's paper at the Symposium on Dynamic Languages 2010 [Bru10b]. This section introduces a simple, yet efficient optimization technique to reduce the interpretation overhead for loading local variables onto the operand stack. First, we present findings of a quantitative analysis (Section 3.5.1). Next, we show how to statically estimate the utility of multiple local variables in the presence of a restricted local cache variable set (Section 3.5.2). Finally, we demonstrate how to implement partial stack frame caching, as well as present an evaluation of the effectiveness of using our static scoring estimate.

3.5.1 Basic Idea

Using dynamic bytecode frequency measurement we can easily break down which instructions are executed most often during the course of executing any given program (cf. Section 5.2.1.) We see that loading operands onto the operand stack constitutes *the* most often executed instruction in the Python interpreter. This has been independently confirmed in other studies as well, for example when comparing the effectiveness of register and stack architectures for the Java virtual machine [SCEG08], they cite a similar figure, i.e., about 42% of all executed instructions are load and store instructions.

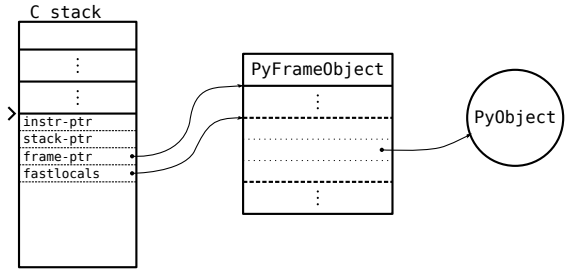
In a previous section (cf. Section 3.1.2) we already detailed on how to efficiently implement the loading of constant data object references, and global data object references. However, the `LOAD_FAST` instruction for loading function arguments and local variable references onto the operand stack is executed more often and can be optimized, too. The Python interpreter implements `LOAD_FAST` the following way:

```
TARGET(LOAD_FAST)
    x = GETLOCAL(oparg);
    if (x != NULL) {
        Py_INCREF(x);
        PUSH(x);
        FAST_DISPATCH();
    }
    ...

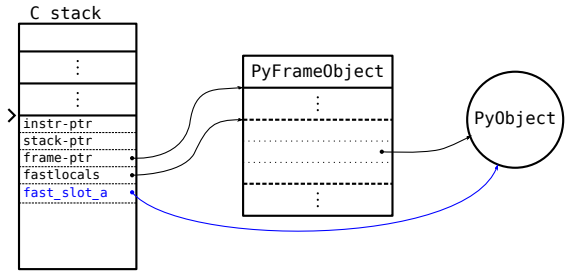
#define GETLOCAL(i) (fastlocals[i])
```

Figure 3.22a shows how `LOAD_FAST` uses an array indirection via `fastlocals` to access local variable references from the heap-allocated Python stack-frame object. `Fastlocals` is kept in the native machine stack frame that holds the interpreter internals, such as the instruction pointer and the stack pointer. All objects to the right of the native machine, or C stack frame, of the interpreter are heap allocated and the Python stack frame object contains a contiguous area of memory that is big enough to manage all local variable and function argument references.

Using such a setup, the idea of partial stack frame caching is simple: extend the interpreter stack frame to directly cache data object references (cf. Figure 3.22b.) This eliminates the array indirection via the `fastlocals` reference as defined by the `GETLOCAL` macro. While the idea is rather simple, we first have



(a) Looking up local variable references using `fastlocals`.



(b) Basic idea of partial stack frame caching.

Figure 3.22: Partial stack frame caching illustrated.

to find out how many of the additional stack frame cache slots (`fast_slot_a`) we need.

3.5.2 Allocating Stack Frame Cache Slots

As we can see in Section 5.2.2, the number of local variables per Python function depends largely on the benchmark program. However, despite the irregularities, we find that a small amount is more frequent than a large amount, i.e., if we have just four local variable cache slots, we can find optimal cache slot allocation for most interpreter executions. In such a case, the optimal cache slot allocation is trivial: because most of the functions have fewer than four local variable references, we just allocate them to the available slots before actually executing any code.

However, the dual, non-trivial, case is interesting, too. Whenever we want to execute a function that has more than just four local variable references, just allocating the first four to the available cache slots is likely to be sub-optimal. Therefore, we present a way to statically estimate which local variables should be promoted to the available cache slots in such a situation.

Whenever we face the situation where we have more local variables than available caching slots, we can either:

- use a heuristic to determine which variable occurrences are likely to benefit most from allocation to caching slots,
- use profiling information to choose the optimal candidates among all local variable occurrences.

Of course, both choices are not mutual exclusive: we can use the heuristic for promoting variables to cache slots before executing, and do profiling to correct incorrect predictions. It turns out, however, that our heuristic predicts the outcomes fairly well (cf. Section 5.2.1.), which is why we only present our implementation of this technique.

The basic argument for estimating success of allocating a variable to a cache slot is its execution frequency. An allocation of local variables to the available cache slots is *optimal*, if it eliminates the most array indirections for any given Python function by replacing its `LOAD_FAST` instructions by instructions using the cache slot instead. Thus, in the trivial case where we have fewer local variables than available cache slots, we can eliminate all `LOAD_FAST` array indirections. In order to estimate the execution frequency, we could simply count all occurrences of local variable load instructions. However, this would discount the fact that certain variable occurrences are likely to be executed more frequently than others: those occurrences within loops. Hence, we use the additional information of loop scopes to weigh occurrences more highly.

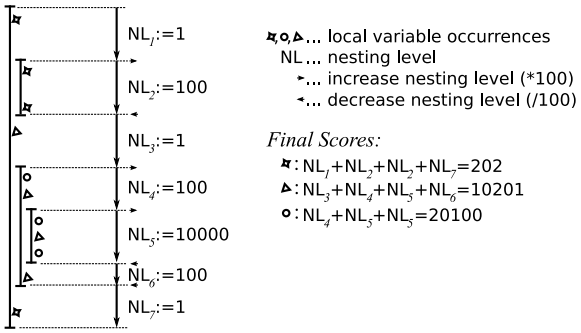


Figure 3.23: Computing the score for local variable occurrences.

Figure 3.23 illustrates how we compute the score for each load local variable instruction occurrence in one pass, i.e., with linear time complexity. The leftmost vertical line conceptually corresponds to a Python function. All parallel vertical lines correspond to loops inside that Python function. Note that while the basic approach might resemble linear scan register allocation to some extent [PS99], partial stack-frame caching does not have any spill instructions.

The remainder of this section explains our implementation of computing the scores, selecting the top n scores for n available cache slots, the derivative instructions using the cache slot variables instead, and finally an evaluation of the effectiveness of this heuristic.

The following listing contains all local variables we need for computing the scores:

```
int open_blocks[16]= {0, 0, 0, 0,
                      0, 0, 0, 0,
                      0, 0, 0, 0,
                      0, 0, 0, 0};
unsigned short open_block_top= 0;

int cur_block_end= -1, scope_depth= 1;
```

```
int max1= -1, max2= -1, max3= -1, nlocals= co->co_nlocals;

unsigned int scoring[nlocals];

unsigned int i= 0, n= Py_SIZE(co->co_code);
bytecode_t instr_ptr;
opcode_t opcode;
int oparg= 0;
```

`Open_blocks` and `open_block_top` are used to maintain a stack of open block endings in the form of instruction offset indexes that close the currently open block. As you can see, we limit ourselves to a maximum of 16 open blocks which corresponds to 16 cascaded loops. `Cur_block_end` contains the instruction offset that closes the current block. `Scope_depth` corresponds to the `NL` nesting level of Figure 3.23, i.e., we multiply/divide our constant block-depth weight (100) with the `scope_depth` variable and add its current value to local variable occurrences we find. In `max1`, `max2`, `max3` we store the top-3 maximum scores of our computation. The enumeration of the variables encodes its invariant, viz. $max1 > max3 > max3$. `Nlocals` contains the number of local variables for a given code object (`co` points to an instance of `PyCodeObject`/`PyCode_Type`). `Scoring` is the array holding all scores for all local variables. For processing all instructions we use the local variables `i` and `n`. `Instr_ptr`, `opcode`, `oparg` are the variables we use for decoding instructions.

Score Calculation

The following C listing contains the code for computing the scores for each local variable. Note that while we could calculate the scores separately using a dedicated pass over the instructions, our implementation integrates the score calculation when we create our regular instruction format, i.e., we combine both into just one pass.

```
while (i < n) {
  /* A: check for end of current block... */
  if (cur_block_end == INSTR_OFFSET()) {
    scope_depth/= 100;
    cur_block_end= open_blocks[--open_block_top];
  } // if

  /* B: decode instr and advance instr ptr... */
  opcode= *instr_ptr++;   i++;
  if (HAS_ARG(opcode)) {
    oparg= instr_ptr[0] << 8 + instr_ptr[1];
    instr_ptr+= 2;        i+= 2;
  } // if

  /* C: increment score for an occurrence of a load/store fast pair... */
  if (opcode == LOAD_FAST || opcode == STORE_FAST) {
    scoring[oparg]+= scope_depth;
  } // if
  /* D: score computation for LOAD_DEREF missing */

  /* E: check if we have another block inside of the current one... */
  if (opcode == SETUP_LOOP) {
    scope_depth*= 100;
    open_blocks[open_block_top++]= cur_block_end;
    cur_block_end= INSTR_OFFSET() + OPARG;
  } // if
```

```
} // while
```

The following steps explain the code in detail:

A: Because the `instr_ptr` always points to the next instruction, we have to check if our current instruction offset closes any currently open block scope. This can not be done after instruction decoding, because the instruction offset does not correspond to the current instruction but to its successor instead. If we close any currently open block, we decrease the current nesting level and pop the next block scope closing offset from the block stack.

B: Decode the instruction pointed to by `instr_ptr` and assign values to `opcode` and `oparg`.

C: Increment the score of a local variable occurrence for the local variable denoted by `oparg`, if the `opcode` is either `LOAD_FAST` or `STORE_FAST`. Therefore, code sequences with frequent `STORE_FAST` instruction occurrences benefit from partial stack frame caching, too.

D: If we want to calculate the scores for other instruction types, such as occurrences of `LOAD_DEREF`, too, we could insert the matching code here.

E: If the current instruction creates another loop block, increment the current nesting level and use the block stack to properly handle cascaded loops.

When we terminate this loop, `scoring` contains all the positive integer scores for all local variables in the range $[0, nlocals)$.

Selecting Top-N Scores

After computing the scores for each local variable reference, we need to choose the top scoring variable references for the number of available cache slots. The following listing shows how we can choose any fixed amount of n maximum scores in just one linear pass, i.e., without needing to sort the list of scores first (where $n := 3$):

```
for (i= 0; i < nlocals; i++) {
  int elem= scoring[ i ];
  if (elem > max1) {
    max3= max2;
    max2= max1;
    max1= elem;
  }
  else if (elem > max2) {
    max3= max2;
    max2= elem;
  }
  else if (elem > max3) {
    max3= elem;
  } // if
} // for
```

We see that by leveraging the transitive nature of the invariant $max1 > max2 > max3$ for the top-most three elements, updates "trickle" down the list of top-most scores.

Quickening Derivatives

Once we know which local variables to promote to their cache slots, we have to quicken the corresponding occurrences to their optimized derivatives. The following listing presents derivative instructions that use the cache slot instead of the array indirection implemented in the `GETLOCAL` macro:

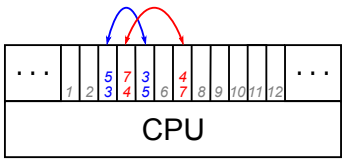
```
TARGET(LOAD_FAST_A)
    x = fast_slot_a;
    if (x != NULL) {
      ...
    }
    ...

TARGET(STORE_FAST_A)
    Py_XDECREF(fast_slot_a);
    fast_slot_a= POP();
    FAST_DISPATCH();
```

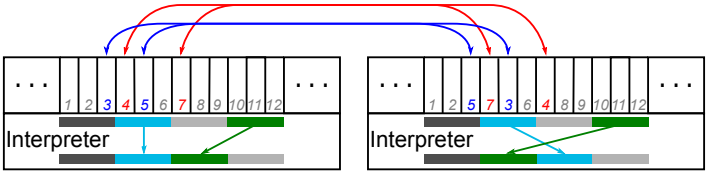
We compute the scores while creating the new instruction format. Thus we can quicken the promoted `LOAD_FAST` instructions to their optimized derivatives in the same pass—otherwise another pass would be necessary.

3.6 Interpreter Instruction Scheduling

Parts of this section will be published at the upcoming International Conference on Compiler Construction, 2011 [Bru11]. Instruction scheduling is a very well known compiler optimization technique [Muc97, Mor98, CT04]. Its objective is to re-order native machine instructions of the input program for optimized execution on the exact same fixed native machine, while retaining the original semantics of the input program. It turns out that for an interpreter, the situation is the exact opposite. Usually, for an interpreter we cannot change the order of the instructions for the input program without changing its semantics. However, we are free to change the interpreter, because it is not fixed as is the native machine for a compiler.



(a) Instruction scheduling in a compiler.



(b) Interpreter instruction scheduling.

Figure 3.24: Instruction scheduling.

Figure 3.24 illustrates this difference. Figure 3.24a shows how instruction scheduling permutes assembly instructions of the input program. The red and blue arrows and permuted instruction numbers denote that the program semantics remain unchanged by instruction scheduling. This is contrary to the illustration in Figure 3.24b: We see that changing the instructions of the interpreter usually changes semantics (red and blue arrows), i.e., changing the instruction order changes the input program. However, we are free to re-arrange the operation implementations inside the interpreter to optimize the interpretation of the input program (shaded rectangles inside the Interpreter rectangle).

Given this setting, interpreter instruction scheduling tries to optimize the interpreter's instruction arrangement in such a way that we optimize instruction cache utilization of the native machine that executes the interpreter. The basic idea is to find particularly frequently executed sequences of interpreter instructions and arrange them to be co-located in memory, such that whenever the instruction dispatch from one instruction to its successor occurs, the instruction cache already contains the operation implementation of the successor instruction.

Interpreter instruction scheduling is particularly effective for interpreters that provide many optimized interpreter instruction derivatives, as is the resulting interpreter when applying the previously presented purely interpretative optimization techniques (cf. Sections 3.3, 3.4, 3.5.)

3.6.1 Formalization

We present a formal description of the problem of interpreter instruction scheduling.

$$
\begin{aligned}
&I := i_0, i_1, \ldots, i_n \\
&A := a_0, a_1, \ldots, a_n \\
&P := p_0, p_1, \ldots, p_m \\
&\quad \forall p \in P : \exists j : i_j \in I \wedge a_j \in A \Leftrightarrow p = (i_j, a_j) \\
&T := \{(p, f) \mid p \in P \wedge f \in \mathbb{N}\} \\
&K := \{p \mid (p, f) \in T \wedge f \geq L\} \\
&K \subset P
\end{aligned}
\tag{3.1}
$$

We define an interpreter I as a set of n instructions i. To each instruction i corresponds a native machine address a of the set of n addresses A, i.e., the address for some interpreter instruction i_j is a_j. Next, we define a program P consisting of m instruction occurrences, which are tuples of an instruction i and the corresponding address a. This concludes the definition of the static view on an interpreter. However, our optimization requires profiling information obtained at run-time. Thus, we define the trace T of a program P as the set of all tuples of an instruction occurrence p and its execution frequency f. Since a trace contains much more information than we need, we define a kernel K, that contains all instruction occurrences p of our program P that have execution frequencies above some definable threshold L.

Given these definitions, the following functions allow us to precisely capture the concept of distance between instructions.

$$
\begin{aligned}
s(p_i) &:= |a_{i+1} - a_i| \\
d(p_i, p_j) &:= \begin{cases} |a_j - a_i| - s(p_i) & \text{if } i \leq j, \\ |a_i - a_j| - s(p_j) & \text{if } i > j. \end{cases} \\
d_{overall}(P) &:= \sum_{j=1}^{m} d(p_{j-1}, p_j)
\end{aligned}
\tag{3.2}
$$

First, we define a function s that computes the size of an instruction i. Next, we define a function d that computes the distance of two arbitrary instructions. Here, the important thing is to note, that if two instruction occurrences p_i and p_j refer to two adjacent instructions, i.e., $p_i = (i_i, a_i)$ and $p_j = (i_{i+1}, a_{i+1})$, then the distance between them is zero. ($d(p_i, p_j) = |a_{i+1} - a_i| - |a_{i+1} - a_i|$) Finally, the overall distance of a program is the sum of all of its sequential distances. Using static program data, this makes no sense, because we do not a priori know which parts of program P are hot. Here, we use our kernel K, which contains only relevant parts of the program, with respect to the overall computational effort. Thus, we define interpreter instruction scheduling as finding a configuration of interpreter instructions that results in a minimal overall distance over some kernel K.

3.6.2 Finding Computational Kernels

Section 3.6.1 already mentions the importance of scheduling instructions only for computational kernels of any given program. Thus, we avoid the computation of instruction schedules for infrequently executed sequences. We collect instruction traces from benchmark programs and use them as an input for the actual instruction scheduling algorithm of Section 3.6.3.

Obtaining Traces

Similarly to the easy way of obtaining dynamic bytecode frequency measurements, we can collect instruction traces from any running benchmark program by the following two-step procedure:

1. Instrumenting the interpreter: Alter the implementation of the instruction dispatch mechanism to print relevant pieces of information to a stream, such as `stdout` or `stderr`.

2. Running the interpreter with the benchmark program and pipe the tracing output to a collector program.

This approach has very nice properties. First of all, it is a very simple procedure that does not involve intricacies. Second, the actual instruction trace is generated and subsequently immediately consumed by our collector program. Since we are only interested in the aggregated data anyways, this saves the more or less expensive round-trip of temporarily saving the data to a—potentially very big—file and processing the files' contents after the fact—which depending on the file size can have its own difficulties. And while none of these things are insurmountable objectives for any self-respecting programmer, we found that this simple approach is ideally suited for all of our analysis tasks, e.g., the same approach and often same collector is used for collecting dynamic bytecode frequency measurements, instruction traces, aggregating the total number of increment and decrement reference count operations.

The following two listings contain our collector program implementation in Python:

```
class Occurrence(object):

    def __init__(self, addr):
        self.addr= addr
        self.instructions= {}
        self.occurrences= {}

    def add(self, offset, identifier):
        if offset not in self.instructions:
            self.instructions[offset]= identifier

        if offset not in self.occurrences:
            self.occurrences[offset]= 0

        if self.instructions[offset] != identifier:
            self.instructions[offset]= identifier

        self.occurrences[offset]+= 1

    def hottness(self):
        return sum(self.occurrences.values())
```

For all Python functions we create a dedicated `Occurrences` object. This object collects all instruction occurrences that belong to it. We distinguish the instruction occurrences by their instruction offset, and store the corresponding instruction identifier in a separate dictionary, i.e., a hash-table.

```
def collect():
    codeobjects= {}
    overall= 0

    for l in sys.stdin:
        try:
            (code_addr, offset, ident, arg)= l.split(',')
            if code_addr not in codeobjects:
                codeobjects[code_addr]= Occurrence(code_addr)

            codeobjects[code_addr].add(int(offset), ident + str(int(arg)))
            overall+= 1
        except:
            pass

    for x in sorted(codeobjects.values(), key=lambda x: x.hottness()):
        print "Occurrence BEGIN", x.hottness(), overall / x.hottness()
        x.show()
        print "Occurrence END"
```

The `collect` procedure does the actual "heavy lifting" of our collector program. The main loop consists of reading lines `l` from `stdin` until we have processed all information generated by the instrumented interpreter. Next, we break apart the input line by the pre-defined delimiter character—in our case a comma "`,`" character. Then, we create an `Occurrence` object if an object corresponding to the current code address (`code_addr`) is not found. Once, we have a corresponding `Occurrence` object, we just add this instruction occurrence to that object.

After we have collected all instruction traces, we print all collected instruction traces grouped by the function that contains all these instructions. For estimating the overall computational impact of any specific code object, we compute the ratio of the computational effort of the code object in question to the overall computational effort.

Example

For further illustration, we introduce a working example here. We are going to take a close look on how interpreter instruction scheduling works, using the `fasta` benchmark of the computer language benchmarks game [Ful]. Running the `fasta` program on the Python interpreter, for example with an argument of 50,000, results in the execution of 10,573,205 interpreter instructions. We extract a complete trace using the method described in the previous section, i.e., with an instrumented interpreter and an aggregating collector program. If we restrict ourselves to only consider kernels for interpreter instruction scheduling, we can significantly reduce the amount of information to consider. For example, an aggregated trace of the `fasta` program shows that the interpreter executes 5,976,237 instructions while interpreting the `genRandom` function, i.e., more than half of the totally executed instructions can be attributed to just one function (cf. Table 3.4.) Another function—an anonymous list comprehension—requires 4,379,824 interpreted instructions (cf. Table 3.5.) Together, the `genRandom` function and the list comprehension constitute 97.95% of all executed instructions.

Frequency	Offset	Instruction Identifier
1	16	STORE_FAST_A
1	24	LOAD_GLOBAL_NORC
1	32	LOAD_FAST_B_NORC
1	40	CALL_FUNCTION_NORC
1	48	STORE_FAST_C
1	56	SETUP_LOOP
396,036	64	LOAD_FAST_A_NORC
400,000	72	LOAD_FAST_NORC
400,000	80	INCA_LONG_MULTIPLY_NORC
396,037	88	LOAD_FAST_NORC
400,000	96	INCA_LONG_ADD_NORC_TOS
396,041	104	LOAD_FAST_B_NORC
400,000	112	INCA_LONG_REMAINDER_NORC_TOS
396,040	120	STORE_FAST_A
400,000	128	LOAD_FAST_D_NORC
400,000	136	LOAD_FAST_A_NORC
400,000	144	INCA_FLOAT_MULTIPLY_NORC
396,039	152	LOAD_FAST_C_NORC
400,000	160	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
396,039	168	YIELD_VALUE
399,999	184	JUMP_ABSOLUTE

Table 3.4: Dynamic bytecode frequency for `genRandom` function of benchmark program `fasta`.

Frequency	Offset	Instruction Identifier
6,600	16	LOAD_FAST_A
402,667	24	FOR_ITER_RANGEITER
396,002	32	STORE_FAST_B
399,960	40	LOAD_DEREF
396,001	48	LOAD_DEREF_NORC
396,000	56	LOAD_DEREF_NORC
396,000	64	LOAD_DEREF_NORC
396,000	72	FAST_PYFUN_DOCALL_ZERO_NORC
395,999	80	FAST_C_VARARGS_TWO_RC_TOS_ONLY
395,999	88	INCA_LIST_SUBSCRIPT
395,998	96	LIST_APPEND
395,998	104	JUMP_ABSOLUTE
6,600	112	RETURN_VALUE

Table 3.5: Dynamic bytecode frequency for an anonymous list comprehension of benchmark program `fasta`.

Though Tables 3.4, and 3.5 indicate that our trace gathering tool is imprecise, since it seems to lose some instruction traces, it is precise enough to indicate which parts of the instructions are kernels. For example, the kernel of function

`genRandom` includes all 15 instructions between the offsets 64 and 184, whereas the kernel of the anonymous list comprehension includes all 12 instructions between the offsets 24 and 104. In consequence, our interpreter instruction scheduling algorithm only has to consider the arrangement of 27 instructions which constitute almost the complete computation effort of the `fasta` benchmark. If all 27 instructions are distinct, the optimal interpreter instruction scheduling consists of these 27 instructions being arranged sequentially and compiled adjacently, according to the order given by the corresponding kernel. However, because of the repetitive nature of load and store instructions for a stack-based architecture, having a large sequence of non-repetitive instructions is highly unlikely. Therefore, our interpreter instruction scheduling algorithm should be able to deal with repeating sub-sequences occurring in a kernel. In fact, our `fasta` example contains repetitions, too. The `genRandom` function:

- `LOAD_FAST_A_NORC`, at offsets: 64, 136.
- `LOAD_FAST_NORC`, at offsets: 72, 88.

The anonymous list comprehension contains the following repetition:

- `LOAD_DEREF_NORC`, at offsets: 48, 56, 64.

Fortunately, however, only single instructions instead of longer sub-sequences repeat. Therefore, for the `fasta` case, an optimal interpreter instruction scheduling can easily be computed. We generate a new optimized instruction set from the existing instruction set and move instructions to the head of the dispatch loop according to the instruction order in the kernels. We maintain a list of all instructions that have already been moved, and whenever we take a new instruction from the kernel sequence, we check whether it is already a member of that remembered list. Thus, we ensure that we do not re-reorder already moved instructions. For our `fasta` example, this means that for all of the repeated instructions, we only generate them when we process them for the first time, i.e., only at the first offset position for all occurrences. Consequently, interpreter instruction scheduling generates chains of maximum length for subsequently processed instruction sequences that correspond extremely well to the major instruction sequences occurring in the `fasta` benchmark.

3.6.3 Scheduling Instructions

The last paragraph of the previous section already explained the objectives and basic technique of interpreter instruction scheduling and mentions the importance of recognizing repeating sub-sequences in computational kernels of obtained instruction traces. In this section, we demonstrate how our algorithm pays attentions to repeating sub-sequences and we illustrate its inner workings by discussing another example.

Repeating Sub-Sequences

For demonstrating the importance of repeating sub-sequences, we use another benchmark from the computer language benchmarks game [Ful], viz. the `nbody` benchmark. Running the `nbody` benchmark with an argument of 50,000 on top of our instrumented Python interpreter for dynamic bytecode frequency analysis

results in the execution of 68,695,970 instructions, of which 99.9% or 68,619,819 instructions are executed in the `advance` function. Its kernel K consists of a trace of 167 instructions, distributed among just 29 distinct instructions. Given equal probability and distribution and considering only sub-sequences of length 1, each of the 29 instructions would be present 5.7586 times within kernel K.

Instruction Identifier	Offset Occurrences	Sum
INCA_LOAD_CONST_NORC	208, **232**, 272, 296, 336, *360*, 488, 528, 576 624, 704, 784, 864, 944, 1024, 1160, 1200, 1256 1296, 1352, 1392	21
INCA_LIST_SUBSCRIPT_NORC	216, **240**, 280, 304, 344, *368*, 536, 584, 640 720, 800, 880, 960, 1040, 1176, 1208, 1272, 1304 1368, 1400	20
LOAD_FAST_NORC	152, 392, 456, 464, 544, 592, 656, 736, 808 816, 896, 976, 1048, 1056, 1184, 1280, 1376	17
LOAD_FAST_A_NORC	200, 264, 328, 520, 616, 696, 776, 1152, 1192 1248, 1288, 1344, 1384	13
INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	688, 768, 848, 928, 1008, 1088, 1240, 1336, 1432	9
INCA_FLOAT_MULTIPLY_NORC	416, 440, 472, 664, 744, 824, 904, 984, 1064	9
ROT_THREE	680, 760, 840, 920, 1000, 1080, 1232, 1328, 1424	9
DUP_TOPX_NORC	632, 712, 792, 872, 952, 1032, 1168, 1264, 1360	9
INCA_FLOAT_ADD	448, 480, 912, 992, 1072, 1224, 1320, 1416	8
LOAD_FAST_B_NORC	224, 288, 352, 568, 856, 936, 1016	7
INCA_FLOAT_SUBTRACT	248, 312, *376*, 672, 752, 832	6
STORE_FAST	136, 384, 512, 560, 608	5
LOAD_FAST_C_NORC	400, 408, 648, 888	4
INCA_FLOAT_MULTIPLY_NORC_SEC	504, 1216, 1312, 1408	4
LOAD_FAST_D_NORC	424, 432, 728, 968	4
JUMP_ABSOLUTE	1096, 1440, 1456	3
STORE_FAST_A	184, 1144	2
INCA_FLOAT_MULTIPLY_NORC_TOS	552, 600	2
SETUP_LOOP	144, 1112	2
FOR_ITER_LISTITER	168, 1136	2
POP_BLOCK	1104, 1448	2
GET_ITER_NORC	160, 1128	2
STORE_FAST_C	256	1
STORE_FAST_B	192	1
STORE_FAST_D	320	1
INCA_UNPACK_TUPLE_TWO	176	1
FOR_ITER_RANGEITER	128	1
INCA_FLOAT_POWER_NORC_TOS	496	1
LOAD_DEREF_NORC	1120	1

Table 3.6: Dynamic bytecode frequencies for kernel in `advance`.

Table 3.6 shows actual distribution of instruction occurrences within kernel K: For example the two top-most repeating instructions (`INCA_LOAD_CONST_NORC`, and `INCA_LIST_SUBSCRIPT_NORC`) occur frequently in pairs. Hence, the simple

procedure of just creating an instruction schedule by processing the instructions according to the "plan" of the actual kernel K will likely result in a sub-optimal schedule, because it does not pay attention to repeating sub-sequences.

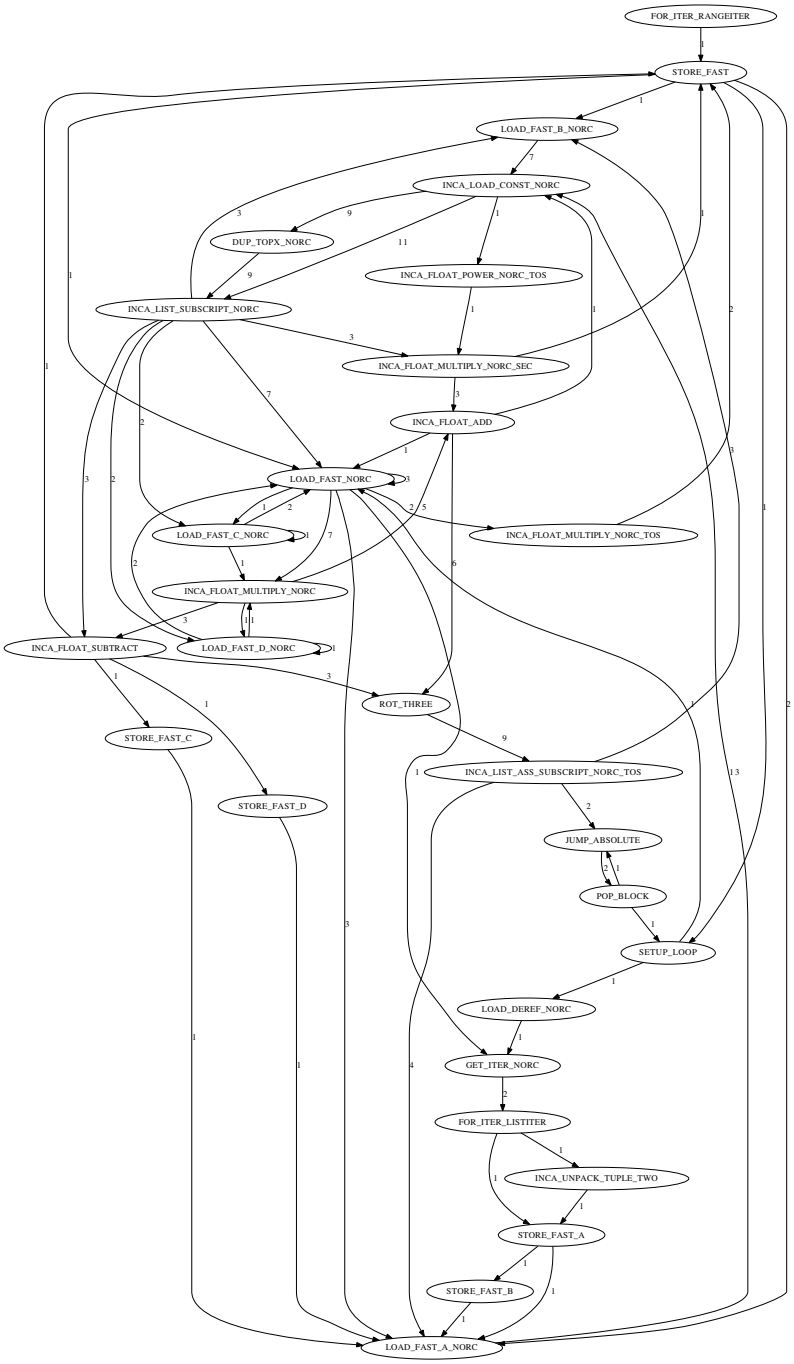


Figure 3.25: Graph from the instructions of the kernel for `nbody` benchmark.

The idea to schedule instructions in the presence of repeating sub-sequences is to create a single-instruction graph from the instructions of the kernel K (cf. Figure 3.25.) However, for every instruction occurrence there is only just one node in the graph, i.e., Figure 3.25 contains just 29 nodes. Whenever we want to add an edge to the graph between a source node s and a destination node d that already exists, we increment the existing edges' weight instead. Consequently, for any given source node s in the resulting graph, we can easily find the most frequent successor node/instruction s' by ordering the edges $s \rightarrow s'$ in descending order according to their weights and choosing the first one. We merge the list of remaining alternate nodes with a global open list. Now, we follow the path from $s' \rightarrow s''$ using the same steps as above until we exhaust the paths. Whenever we cannot find a successor node, we start taking the node with the highest weight from the global open list and start traversing down the most frequent path again. Eventually, we will have processed all nodes from the original path.

Scheduling Algorithm

The following listing shows our implementation in Python, followed by a detailed description of how it works:

```
def rsorted(dict): ## sort dict by reverse order
    return sorted(dict.items(), key=lambda (key, value): -value)

def schedule_instr(graph):
    schedule= []

    open= rsorted(graph.most_frequent_vertices())
    ## open is a list of tuples (node, number of edges)
    open= [ (node, 0) for (node, edge_count) in open ]
    ## now, we have erased the number of edges, such that when
    ## we add the reachable destination nodes for the current
    ## source node, and we sort the <open> list, the node that
    ## can be reached with the edge having the highest weight
    ## will be the first element on the <open> list

    while open:
        ## fetch the tuple, ignore the number of edges
        (n, _)= open.pop(0)
        while n:
            if n not in schedule:
                schedule.append( n )

            reachable= rsorted(n.get_destinations())
            if not reachable: break

            ## find reachable nodes that have not been scheduled yet
            (m, _)= reachable.pop(0)
            while m in schedule:
                if len(reachable) > 0:
                    (m, _)= reachable.pop(0)
                else:
                    m= None

            if m:
                n= m                                ## assign successor node
                open= rsorted( reachable + open ) ## keep reachable nodes sorted
            else: break

    return schedule
```

Note that instead of using a simple list and enforcing a certain order by repeatedly calling the `rsorted` function on that list, we could very well use a priority queue. However, we decided not to use a priority queue, because its standard Python implementation is tied to the implementation heap queue algorithm (for example the corresponding module name is `heapq`.) This considerably complicates the previous algorithm by introducing an implementation detail which does not improve the illustration of the algorithm's mechanics.

The algorithm requires the following steps. First, we create a list named `open` that contains tuples of nodes and the collective weight of edges leading to that node. We sort the `open` list in descending order of the collective weight component. Because we actually only need the collective weight for choosing the first node and ensuring that we process all nodes, we can now safely zero out all weights of the tuples in the `open` list. Then, we start the actual algorithm by fetching and removing the first tuple element from the `open` list; we assign the node part to n and ignore the weight. Next, we check whether n has already been scheduled by checking whether the `schedule` list contains n. If it has not been scheduled yet, we append it to the `schedule` list. Then, we start looking for a successor node m. We process the successor nodes by having them sorted in descending order of the edge-weight associated between nodes n and m. We repeatedly fetch nodes m from the list of successors until we find a node that has not already been scheduled or the list is finally empty. If we do not find a node m, then we have to restart by fetching the next node from the `open` list. If we find a node m, then we add the reachable nodes from n to m to the `open` list and sort it, such that the successors with the highest weight will be chosen as early as possible. Next we assign m to n and restart looking for m's successors.

Table 3.7 contains the schedule that our algorithm generates for the kernel shown in Figure 3.25 and Table 3.6.

No.	Instruction	No.	Instruction
1	INCA_LOAD_CONST_NORC	16	LOAD_DEREF_NORC
2	INCA_LIST_SUBSCRIPT_NORC	17	GET_ITER_NORC
3	LOAD_FAST_NORC	18	FOR_ITER_LISTITER
4	INCA_FLOAT_MULTIPLY_NORC	19	STORE_FAST_A
5	INCA_FLOAT_ADD	20	STORE_FAST_B
6	ROT_THREE	21	STORE_FAST_D
7	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	22	INCA_FLOAT_MULTIPLY_NORC_SEC
8	LOAD_FAST_A_NORC	23	JUMP_ABSOLUTE
9	DUP_TOPX_NORC	24	POP_BLOCK
10	LOAD_FAST_B_NORC	25	LOAD_FAST_C_NORC
11	INCA_FLOAT_SUBTRACT	26	LOAD_FAST_D_NORC
12	STORE_FAST_C	27	INCA_UNPACK_TUPLE_TWO
13	INCA_FLOAT_MULTIPLY_NORC_TOS	28	INCA_FLOAT_POWER_NORC_TOS
14	STORE_FAST	29	FOR_ITER_RANGEITER
15	SETUP_LOOP		

Table 3.7: Interpreter Instruction Schedule for the `nbody` benchmark.

3.6.4 Compiling Instruction Schedules

Once we have computed a schedule of interpreter instructions, we need to compile the interpreter with that schedule. We have extended our interpreter generator from our previous work ([Bru10b]) to generate all instructions, not just the optimized derivatives. Since we already have a schedule, it is straightforward to generate an optimized instruction set from the standard instruction set. We just process the schedule in order, move instructions from the old instruction set, and add these instructions to the optimized instruction set. Once, we have processed the plan, we just add the remaining instructions to the new optimized instruction set.

There are compiler optimizations that can change the instruction order as computed by our interpreter instruction scheduling. First of all, basic block reordering as done by `gcc` 4.4.3 will eliminate our efforts by reordering basic blocks after a strategy called "software trace-cache" [RLPN+99]. Fortunately, we can switch this optimization off, by compiling the source file that contains the interpreter dispatch routine with the additional flag `-fno-reorder-blocks`. However, the instructions are still entangled in a switch-case statement. Since it is possible for a compiler to re-arrange case statements, we decided to remove the switch-case statement from the interpreter dispatch routine as well. Because our interpreter is already using the optimized threaded code dispatch technique [Bel73], removing the switch-case statement is simple. However, we stumbled upon a minor mishap: `gcc` 4.4.3 decides to generate two jumps for every instruction dispatch. Because the actual instruction-dispatch indirect-branch instruction is shared by all interpreter instruction implementations, available expression analysis indicates that it is probably best to generate a direct jump instruction back to the top of the dispatch loop, directly followed by an indirect branch to the next instruction. On an Intel Nehalem (i7-920), `gcc` 4.4.3 generates the following code at the top of the dispatch loop:

```
.L1026:
    xorl %eax, %eax
.L1023:
    jmp  *%rdx
```

And a branch back to the label `.L1026` at the end of every instruction:

```
    movq opcode_targets.14198(,%rax,8), %rdx
    jmp  .L1026
```

Of course, this has detrimental effects on the performance of our interpreter. Therefore, we use `gcc`'s `-save-temps` switch while compiling the interpreter routine with `-fno-reorder-blocks` to retrieve the final assembler code emitted by the compiler. We implemented a small fix-up program that rebuilds the basic blocks and indexes their labels from the interpreter's dispatch routine (`PyEval_EvalFrameEx`), determines if jumps are transitive, i.e., to some basic-block that itself contains only a jump instruction, and copies the intermediate block over the initial jump instruction. Thus, by using this fix-up program, we obtain the original threaded-code jump sequence:

```
    movq opcode_targets.14198(,%rax,8), %rdx
    xorl %eax, %eax
    jmp  *%rdx
```

Finally, we need to assemble the fixed-up file into the corresponding object file and link it into the interpreter executable.

Compiling Instruction Schedules for the PowerPC 970

Our compiler on the PowerPC 970 system (`gcc` version 4.3.2) generates the same code as the for the Intel Nehalem i7-920, i.e., a transitive intermediate jump back to the head of the loop before actually dispatching to the next instruction. Porting our fix-up program was actually very easy, we had to change some regular expressions detecting branches and returns specific to the PowerPC 970 assembly. Unfortunately, however, the principal approach of using a fix-up program to create dedicated instruction schedules has a major draw-back for RISC architectures: Whenever we replace a single branch or jump instruction back to the top of the dispatch loop with the sequence of assembly instructions that perform the actual dispatch, the operation implementation of instructions inevitably grows. Since the fixed instruction-set architecture of a RISC CPU can only use a fixed amount of offsets for branches (in the PowerPC 970, this maximum width is 16 bits), we simply cannot optimize the complete interpreter.

Since we did not have time to come up with a proper fix-up procedure in time for publication of this thesis, we had to compromise. Depending on the actual benchmark, and its corresponding instruction schedule, we had to limit ourselves to a maximum amount of operation implementations to fix-up.

Benchmark	Max. Fix-ups
`binarytrees`	44
`fannkuch`	39
`fasta`	15
`mandelbrot`	39
`nbody`	39
`spectralnorm`	39

Table 3.8: Maximum number of operation implementation fix-ups per benchmark.

Table 3.8 contains our choice of maximum possible fix-ups per benchmark. For the `fasta` benchmark, we had to limit ourselves to a maximum number of 15 operation implementation fix-ups until `gcc` complained about an invalid instruction argument outside of the 16 bit boundary. While it seems that this has detrimental effects on the performance, our in-depth discussion in the evaluation chapter (cf. Section 5.4) convincingly shows that this does not affect performance to a big extent. For future work, however, we are very interested in changing our fix-up routine to implement the optimization for all 394 interpreter instructions.

3.7 Code Generator

To describe the original operation implementation for all of our instructions, we use the Mako template string substitution system [Bay10]. We chose the Mako template system because it is easy to install and provides the functionality we need, i.e., conditionals and iteration constructs. The following template is an example of how we can generate C code—in this case for an optimized derivative as is used by partial stack frame caching (cf. Section 3.5)—using the Mako template language:

```
TARGET(LOAD_FAST_${ instr.id.upper() })
  Py_INCREF(fast_slot_${ instr.id.lower() });
  PUSH(fast_slot_${ instr.id.lower() });
  FAST_DISPATCH();
```

The sequence `${ ... }` marks a string substitution, and we see that we use the `id` field of the `instr` object to get a matching identifier for this instruction.

Type	Context		
	Scalar	List	Map
PyLong_Type	x		
PyFloat_Type	x		
PyComplex_Type	x		
PyBool_Type	x		
PyUnicode_Type	x	x	x
PyByteArray_Type		x	x
PyDict_Type			x
PyList_Type		x	
PyMap_Type			x
PyTuple_Type		x	x
PySet_Type		x	

Table 3.9: Types with context-dependent functions.

The types in Table 3.9 represent the most basic primitives of the Python language, i.e., they are not defined in modules of the standard library but represent the "core" of the language. Depending on the programs to be run in the interpreter, providing different instruction derivatives might be beneficial; without statistical evidence of real-world usage, however, it seems only natural to promote the most basic primitives to their dedicated instructions. In order to select which operations to implement, it is necessary to know which type implements which operations. One approach would be to parse the C code containing the type definitions and their bindings for several operations, represented as C `struct`'s. While certainly possible, our approach is much simpler: using the `gdb` [SPS09] debugger, one can inspect and print C data structures at runtime.

```
1 = {                                   'PyFloat_Type' : {
  tp_name = 0x5438f6 "float",               'tp_name' : 0x5438f6,
  tp_itemsize = 0,                          'tp_itemsize' : 0,
  tp_dealloc = 0x4ea940 <float_dealloc>,    'tp_dealloc' : 0x4ea940,
  tp_repr = 0x4ec630 <float_repr>,          'tp_repr' : 0x4ec630,
  tp_as_number = 0x799c40,                  'tp_as_number' : {
  tp_as_sequence = 0x0,                         'nb_add' : 0x4edcb0,
  tp_as_mapping = 0x0,                      ...
  ...
```

Figure 3.26: Example of the gdb output on the left side, and the corresponding Python data structure definition on the right side.

The format `gdb` uses when printing this information is already very close to Python data structure definitions. For example, Figure 3.26 displays the

output of `gdb`'s print command with the Python floating point type structure (`PyFloat_Type`) as its argument on the left side, and the corresponding Python data structure definition on the right side. As we can see, converting `gdb`'s runtime data structure output into valid Python data structure definitions is trivial—it is possible to do this in your editor of choice with nothing more than the regular search and replace feature, which is certainly faster than writing a C parser. We put this information into a dedicated Python module, such that further changes, refinements, and additions do not affect the code generator. We captured `gdb` output for multiple types; all in all our type structure master data file has 1700 lines of code. Using this module as a database, the code generator automatically decides which operations to promote to their own instructions and how to properly name them. Whereas the former is necessary to prohibit the naive generation of operation implementations for unimplemented type functions, the latter is necessary for debugging convenience.

The generation of the typed instruction derivatives for inline caching is considerably simpler and more elegant if we use a template engine for string substitution. In fact, it is so easy that we moved the generation of the optimized call instructions to use templates, too—which supports our claim that they are only different by type of call target, but remain predictable and largely constant with varying amounts of arguments.

To implement the elimination of reference count operations, we create dedicated Python classes that inherit the implementation of their parent class. All classes eliminating reference count operations have a common parent class, called `RefcntDerivative`, which supplies all features necessary to implement simple and straightforward automatic removal of `Py_INCREF` and `Py_DECREF` operations. Its major functionality is the enumeration of operands in exactly the same order as they are fetched from the stack—using a simple regular expression. In addition, it provides methods for removing increment and decrement reference count operations when given the variable identifier of the reference count operation to be removed. So, classes realizing the separate optimization strategies are easy to implement; see for example the implementation for the elimination of all reference count operations:

```
class NoRefcnts(RefcntDerivative):

  def generate(self):
    super(NoRefcnts, self).generate()
    ...
    for var_id in self.enumerate_stack_ops():
      self.erase_decrefs(var_id)
```

Our common parent class takes care of some additional work, such as ensuring that whenever we have an optimized function/method call derivative instruction with an explicit `Py_DECREF` operation of the arguments tuple, the common `generate` method of the `RefcountDerivative` replaces its occurrence with a call to our custom non-recursive-freeing de-allocation routine. Rewriting the cache miss `goto` instructions to stay within the same class of reference count operation elimination is another important common task.

Using inheritance for modeling our core "derivative" relationship seems natural and enables us to quickly create the code generation system. Besides the actual operation implementation, we use the generator to generate the actual quickening functions necessary for our two-step quickening scheme:

1. from Python bytecode to reference count operation derivatives,
2. from reference count operation derivatives to inline cached derivatives.

Finally, the code generator generates several auxiliary files, such as the opcode mnemonic definitions, the threaded-code dispatch table, and a debugging function that displays a padded string identifier for an instruction opcode.

3.7.1 Architecture

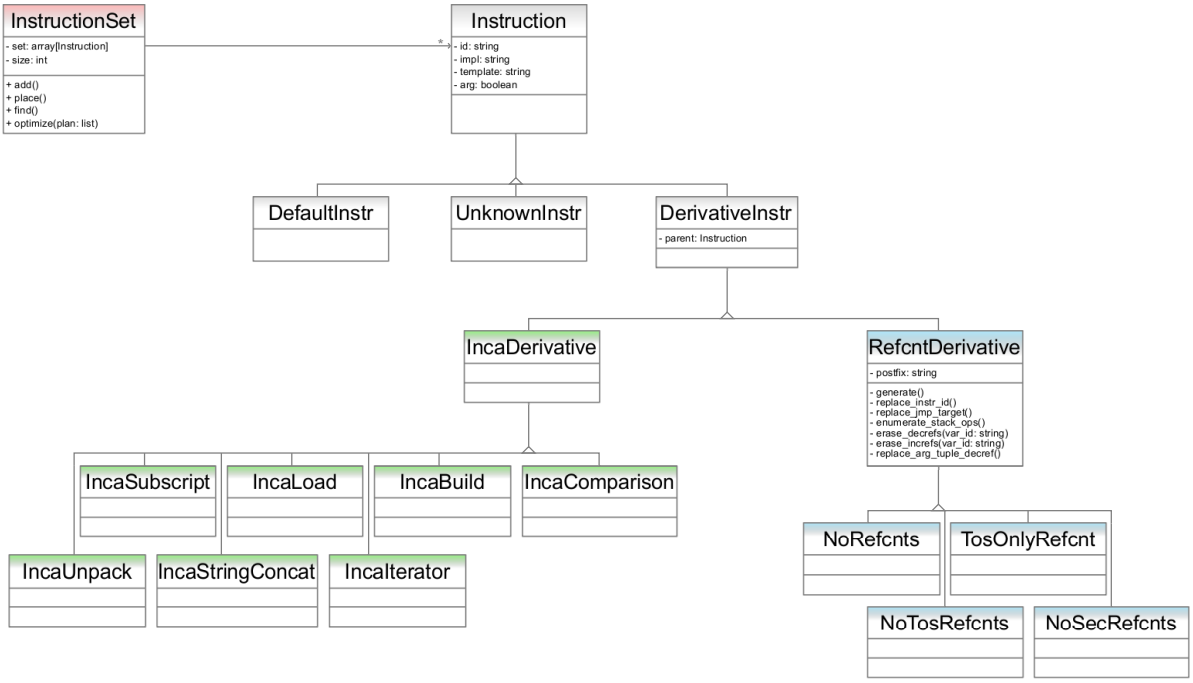


Figure 3.27: UML class diagram of instruction hierarchy.

Figure 3.27 shows a UML class diagram of the code generator's instruction hierarchy. An InstructionSet[2] contains a number of Instruction's. Initially, all instructions are instances of UnknownInstr's, where we add instances of DefaultInstr's when we create the default instruction set. We then use several modules to add derivative instructions to our InstructionSet instance (cf. Figure 3.28). The common base class for all derivatives, DerivativeInstr, has a reference to its parent instruction of which it derives. Thus, we can cascade derivatives, such as having a NoRefcnts derivative of an IncaComparison derivative.

Figure 3.28 shows the pipeline of steps in our code generator. The shaded rectangles are modules that can be enabled and disabled individually to configure the resulting interpreter. With the exception of the Interpreter Instruction Scheduling module, each of these modules creates instruction derivative instances and adds them to the InstructionSet instance that is passed along the pipeline. At

[2]We use the sans-serif font to refer to the diagram.

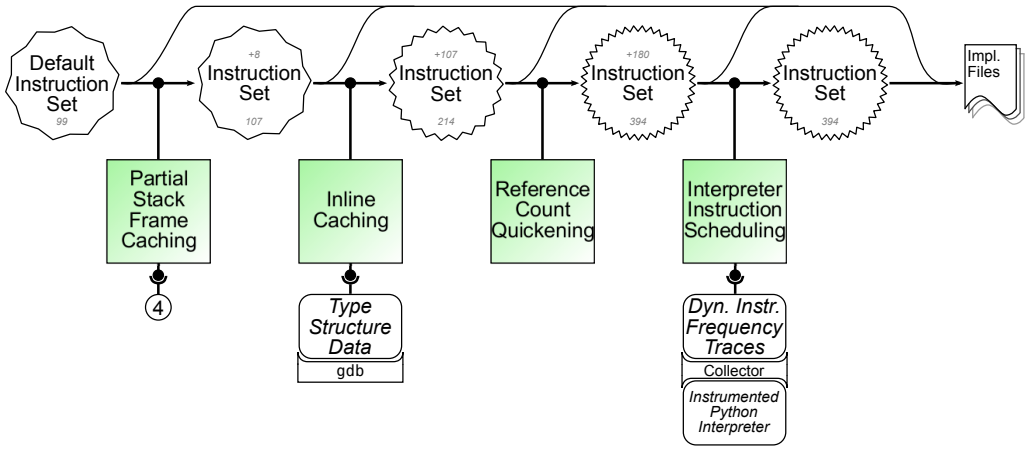


Figure 3.28: Flow diagram of instruction set generation.

every step, we can quit processing the pipeline and directly go to the last step of generating the implementation files. Below the modules, we see necessary input data for each module. For example, the Partial Stack Frame Caching module needs to know how many cache slots to generate code for. Above the modules, we see how each module changes the input InstructionSet-instance. The star-like circle around the labels approximates the amount of instructions in the instruction-set. Thus, we visualize the extent of change by each module. The gray number above the instruction set label displays the number of instructions added to it by the preceding module. The gray number below the instruction set label shows the amount of instructions present in the instruction set. Table 3.10 contains the tabular data visualized in Figure 3.28. The lines-of-code figures given contain empty lines as well and are therefore not exact. The next section discusses the numerical aspects of the code generator.

3.7.2 Implementation in Numbers

We measured the lines of code using the `sloccount` program of David Wheeler [Whe10]. Our code generator produces 6178 lines of C code that is included in the main interpreter. The Python code of our generator amounts to 2739 lines of code, however, out of those 2739 lines of code, 1700 lines of code are consumed by our type-data file generated by raw-data from `gdb`. Therefore, the actual amount of Python code without the master data needed to generate the C code is 1039 lines of code. In addition to the generator and its product, we have manually coded 1759 lines of code. These are 400 lines of code for quickening the `CALL_FUNCTION` instruction and supplying our own version of `unicode_concatenate`, 347 lines of code for our simple abstract interpreter to rewrite the reference count operations, 272 lines of code for the creating and manipulating the new instruction format (including the scoring heuristic), and 87 lines of code that implements the load cache that we described in Section 3.1.2. The remaining lines of code are mostly externalized interpreter macros from the original dispatch loop and smaller auxiliary files.

Type	No. of Instructions	Lines of Code
DefaultInstr	99	1,824
LoadFastStackFrame	4	32
StoreFastStackFrame	4	32
IncaBuild	4	62
IncaComparison	10	200
IncaIterator	15	255
IncaLoad	4	56
IncaOperator	49	1,058
IncaStringConcat	1	18
IncaSubscript	3	57
IncaUnpack	4	76
FastCFunction	2	32
FastCVarArgs	4	155
FastGenerator	1	48
FastPyFunction	3	134
FastPyFunctionDoCall	3	146
FastPyMethod	4	225
NoRefcounts	94	2,095
NoSecRefcounts	12	301
NoTosRefcounts	61	1,324
TosOnlyRefcount	13	484

Table 3.10: Break-down of instructions generated.

Using ‘`cat * | wc -l`‘ to calculate the number of C-code markup inside the Mako templates adds another 1385 lines of code. Here, we cannot use the `sloccount` program, since it does not process files of the Mako template language.

Chapter 4

Related Work

4.1 Purely Interpretative Inline Caching

In his PhD thesis of 1994 [Höl94], Hölzle mentions the basic idea of the data structure underlying our purely interpretative inline caching technique based on interleaving words (cf. Section 3.3.3). The major difference is that we are not only proposing to use this data layout for Self's `send` instruction—or `CALL_FUNCTION` instruction in Python's case—but for all instructions, since there is enough caching potential in Python to justify that decision. Hölzle addresses the additional memory consumption issue, too. We use a simple low-overhead invocation based counter heuristic to determine when to apply this representation, i.e., it is only created for code we know is *hot*. Therefore, we argue that the increased memory consumption is negligible—particularly when compared with the memory consumption of state-of-the-art just-in-time compilers. Section 3.3.3 contains a detailed historical perspective.

In 2007, Haupt et al. [HHD07] published a position paper describing details of adding inline caching to bytecode interpreters, specifically the Squeak interpreter. Their approach consists of adding dedicated inline caching slots to the activation record, similar to dealing with local variables in Python or the constant pool in Java. In addition to a one-element inline cache, they also describe an elegant object-oriented extension that enables a purely interpretative solution to polymorphic inline caches [HCU91]. The major difference to our approach lies in the relative efficiencies of the techniques: whereas our techniques are tightly interwoven with the interpreter infrastructure promising efficient execution, their technique relies on less efficient target address look-up in the stack frame.

Regarding the use of look-up caches in purely interpretative systems, we refer to an article [CPL82] detailing various concerns of look-up caches, including efficiency of hashing functions, etc., which can be found in "Smalltalk-80: Bits of History, Words of Advice" [Kra84]. Kiczales and Rodriguez describe the use of per-function hash-tables in a portable version of common lisp (PCL), which may provide higher efficiency than single global hash tables [KR90]. The major difference to our work is that our inline cache does not require the additional look-up and maintenance costs of hash-tables. Our quickening based approach of Section 3.3.4 eliminates the use of indirect branches for calling inline cached methods, too.

Lindholm and Yellin [LY96] provide details regarding the use of quick instructions in the Java virtual machine. Casey et al. [CEG05] describe details of quickening, superinstructions and replication. The latter technical report provides interesting figures on the performance of those techniques in a Java virtual machine implementation.

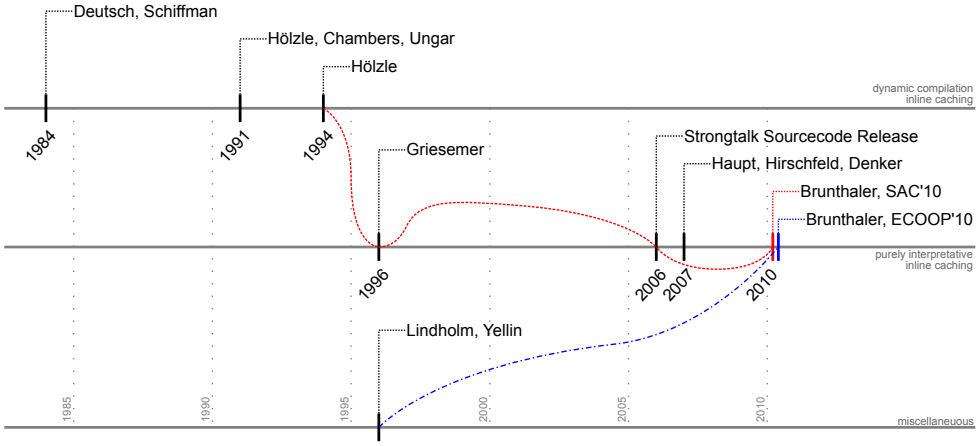


Figure 4.1: Time line for inline caching.

Figure 4.1 shows the time line of the relevant techniques. The red dashed line shows the genesis and rediscovery of the interleaved inline caching technique of Section 3.3.3, including the original Strongtalk implementation of Robert Griesemer in 1996, Sun's decision to open the source of the Strongtalk system, as well as the author's independent re-discovery of the technique in 2009 with the corresponding publication in 2010.

4.2 Reference Count Quickening

Regarding our second contribution, the elimination of reference count operations, we cite the following related work. Introduced by Collins in 1960 [Col60], Deutsch and Bobrow found in 1976 [DB76] that while reference counting has its advantages, the amount of reference count operations caused by local stack modifications, i.e., load and store operations, have a considerable negative impact on the performance of such systems. Hence, Deutsch and Bobrow suggest to remove the immediate processing of reference count operations from the mutator and defer them to a dedicated processing phase—similar to the explicit garbage collection phase of other automatic memory management techniques. Because of their introduction of deferred reference counting, the original reference counting approach is often described as immediate or non-deferred reference counting. As early as 1977, just a year after the deferred reference counting approach described by Deutsch and Bobrow [DB76], Barth described a technique to eliminate reference count operations using a global data-flow analysis in a compiler [Bar77]. In addition to what we describe, Barth's description is able to eliminate more reference count operations than our approach. While our approach works for stack-based interpreters, Barth's description optimizes a derivative

of Pascal that uses reference counting for automatic memory management. Unfortunately, he does not give any evaluation we could use for comparison purposes. Much of the following research on optimizing reference counting focuses on deferred reference counting as suggested by Deutsch and Bobrow [DB76]. Ungar and Patterson [UP82] describe a set of optimization techniques to eliminate redundant reference count operations from the implementation of standard Smalltalk instructions, such as eliminating an increment and decrement reference count operation by directly copying a value from the callee stack frame to the caller stack frame and nilling out the source. These optimization techniques are static and do not take dynamic instruction sequences into account, which is precisely what allows us to eliminate large amounts of reference count operations. As recently as 2006, however, Joisha took up the basic idea of Barth—with much more comprehensive goals [Joi06]. The basic idea is to use data-flow analysis to optimize a research version of a C# compiler that generates code with reference counting for automatic memory management. Joisha uses liveness properties of objects to remove way more reference count operations than our simple approach is able to recognize. His work addresses the "coalescing" of reference count operations that basically corresponds to our approach—but it is only a minor part in his work. His subsequent work of 2008 describes ways to eliminate reference count operations in the presence of modern object-oriented constructs, such as exceptions [Joi08]. While his work achieves a much higher elimination rate of reference count operations, it is certainly not easily realizable in our setting. Our approach does not require any kind of data flow analysis or fix-point computation, but on the other hand can not possibly eliminate as many reference count operations.

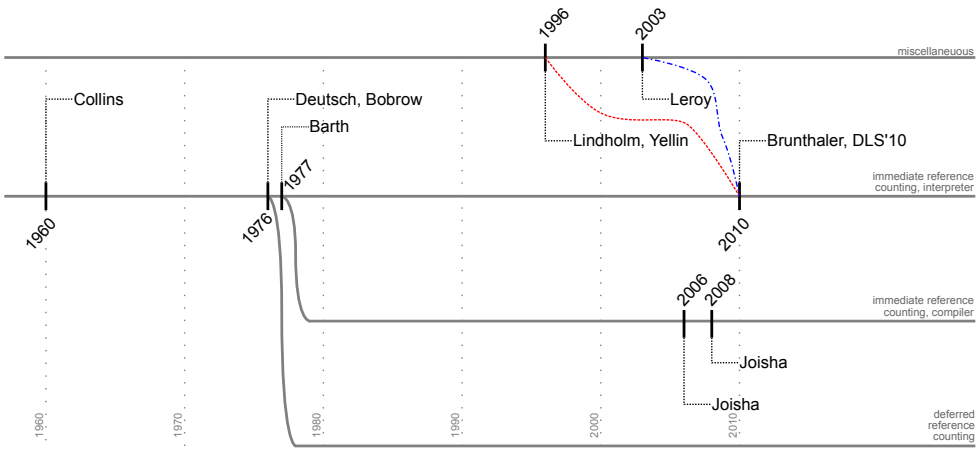


Figure 4.2: Time line for reference count quickening.

Figure 4.2 contains the relevant time line illustrating the corresponding contributions. The figure shows how Deutsch and Bobrow created a dedicated sub-field by deferred reference counting [DB76], and so did Barth in 1977 [Bar77] by the use of static analysis in a compilation setting, which was quite successfully continued by Pramod Joisha in 2006. In addition to the leveraging of 1996's quickening [LY96], we use a simple abstract interpreter over the amount of

reference count operations within a given operation implementation, inspired by the account of Java bytecode verification by Xavier Leroy in 2003 [Ler03].

4.3 Interpreter Instruction Scheduling

Pettis and Hansen [PH90] present their work on optimizing compilers for the Hewlett Packard's PA-RISC architecture. They optimize the arrangement of procedures and basic blocks based on previously obtained profiling information. Interestingly, our reordering algorithm is almost identical to their "algo1" algorithm; they may even be identical, but because no implementation is given, this remains unclear. Another interesting fact is that both our maximum achievable speedups are identical, i.e., both our work achieves a maximum speedup by a factor of 1.14.

More recently, Zhao and Amaral [ZA05] demonstrate algorithms to optimize switch-case computation as well as case-statement ordering in the Open Research Compiler [ope10]. While both our approaches employ information gathered at run-time, the application scenario is quite different. For instance, their approach focuses on optimizing switch-case statements, and they calculate the order in which they should be generated by their rank according to frequency. In contrast, our work focuses on optimization of interpreters, particularly those without using the switch-case dispatch technique. Because of better instruction cache utilization, we choose to use another algorithm that recognizes the importance of properly covering instruction sequences. So in a sense, the major difference is that their optimization approach focuses on larger compiled programs that use switch-case statements, whereas we recognize the nature of an interpreter, where execution remains within its instruction set *at all times*. Another direct consequence of this fundamental difference is that in an interpreter we are usually not interested in the default case, since this indicates an error, i.e., an unknown opcode, which in practice happens never—the exception being malicious intent of a third party.

As for related work on interpreters, the most important work is by Lin and Chen [LC08]. Their work is similar to ours, since they show how to partition interpreter instructions to optimally fit into NAND flash pages. Furthermore, they describe that they too use profiling information to decide which combination of interpreter instructions to co-locate on one specific flash page. Their partitioning algorithm pays attention to the additional constraint of NAND flash page size, i.e., their algorithm computes a configuration of interpreter instructions that fits optimally within the flash pages and keeps dependencies between the pages at a minimum. For the context of our work it is unnecessary to superimpose such a constraint to our algorithm. Though, if one were to set the parameter N determining the NAND flash page size of their algorithm to the maximum representable value, all instructions would be packed into just one partition. Then, our algorithms should produce similar interpreter instruction arrangements. Another difference between our respective approaches is that ours operates on a higher level. While they post-process the assembly output generated by `gcc` to enable their optimizations, our approach is based on re-arranging the instruction at the source code level. Though we admittedly have to fix-up the generated assembly file as well, because of detrimental effects of a misguided optimization. Because of their ties to embedded applications of the technique and its presentation

in that context, we think that our presentation is more general in nature. In addition, we complement our work with extensive performance measurements on contemporary non-embedded architectures.

Ertl and Gregg [EG03a] present an in-depth discussion of two interpreter optimization techniques—superinstructions and replication—to improve the branch prediction accuracy and instruction cache utilization of virtual machines. While the optimization technique of replication is not directly related to interpreter instruction scheduling, it improves the instruction cache behavior of an interpreter at the expense of additional memory. The idea of superinstructions is to combine several interpreter instructions into one superinstruction, thus eliminating the instruction dispatch overhead between the single constituents. While this improves branch prediction accuracy, it improves the instruction cache utilization, too: Since all instruction implementations must be copied into one superinstruction, their implementations must be adjacent, i.e., co-located in memory, which is optimal with respect to instruction cache utilization and therefore results in extremely good speedups of up to 2.45 over a threaded-code interpreter without superinstructions. However, superinstructions can only be used at the expense of additional memory, too. Since interpreter instruction scheduling happens at pre-compile, and compile time respectively, of the interpreter, there are no additional memory requirements—with the notable exception of minor changes because of alignment issues. Because the techniques are not mutually exclusive, using interpreter instruction scheduling in combination with static superinstructions will further improve the performance of the resulting interpreter.

Summing up, the major difference between the related work on compilers and our work is that the former focuses on optimizing elements visible to the compiler, such as procedures, basic blocks, and switch-case statements, whereas our work focuses on re-arranging interpreter instructions—which are transparent to compilers. Related work on interpreters achieves a significantly higher speedup, however, at the expense of additional memory. Our work demonstrates that is possible to improve interpretation speed without sacrificing memory.

Chapter 5

Evaluation

5.1 System Setup and Configurations

We used several benchmarks from the computer language shootout game [Ful]. Since the adoption of Python 3.x is rather slow in the community, we cannot give more suitable benchmarks of well known Python applications, such as Zope, and twisted. However, we used a patch by Martin von Löwis [vL10] for a specific version of `django` to run Unladen Swallow's [unl10] `django` benchmark for measuring a real world web application framework [dja11]. In addition to the `django` benchmark of Unladen Swallow, we provide our extensive analysis data for its `ai` benchmark, consisting of solving the n-queens problem and generating permutations in pure Python code. We ran our benchmarks on the following system configurations:

- Intel i7 920 with 2.6 GHz, running Linux 2.6.32-27 and `gcc` version 4.4.3. (Please note that we have turned off Intel's Turbo Boost Technology to have a common hardware baseline performance without the additional variances immanently introduced by it [Int08].)
- Intel Atom N270 with 1.6 GHz, running Linux 2.6.28-18 and `gcc` version 4.3.3.
- IBM PowerPC 970 with 2.0 GHz, running Linux 2.6.26-2 and `gcc` version 4.3.2.

We used a modified version of the `nanobench` program of the computer language shootout game [Ful] to measure the running times of each benchmark program. The `nanobench` program uses the UNIX `getrusage` system call to collect usage data, e.g., the elapsed user and system times as well as memory usage of a process. We use the sum of both timing results, i.e., elapsed user and system time as the basis for our benchmarks. In order to account for proper measurement and cache effects, we ran each program 50 successive times and the reported data represent arithmetic averages over those repetitions.

5.2 Evaluation of Optimization Potential

5.2.1 Dynamic Bytecode Frequencies

No.	Standard Interpreter	Frequency	Optimized Interpreter	Frequency
1	LOAD_FAST	214,298,992	INCA_LOAD_CONST_NORC	68,845,856
2	LOAD_CONST	90,660,749	LOAD_FAST_A_NORC	61,069,511
3	STORE_FAST	71,733,904	LOAD_FAST_B_NORC	39,059,617
4	BINARY_ADD	45,947,498	LOAD_FAST_NORC	34,729,268
5	CALL_FUNCTION	32,723,786	LOAD_FAST_C_NORC	28,315,920
6	BINARY_MULTIPLY	29,858,698	POP_JUMP_IF_FALSE	24,426,977
7	POP_JUMP_IF_FALSE	27,079,748	RETURN_VALUE	20,739,803
8	COMPARE_OP	23,002,313	INCA_LOAD_CONST	20,710,179
9	RETURN_VALUE	20,739,774	LOAD_FAST_D_NORC	19,437,445
10	LOAD_GLOBAL	18,957,775	STORE_FAST_C	17,296,437
11	BINARY_SUBSCR	17,783,590	INCA_LONG_ADD_NORC	16,806,475
12	FOR_ITER	17,529,343	STORE_FAST_A	15,035,063
13	JUMP_ABSOLUTE	13,466,488	INCA_LOAD_GLOBAL_NORC	14,504,251
14	BINARY_TRUE_DIVIDE	13,450,526	LOAD_FAST_A	14,232,730
15	UNPACK_SEQUENCE	13,390,066	STORE_FAST_B	14,060,314
No. of executed instructions		726,370,483		726,376,463

Table 5.1: Overall comparative dynamic instruction frequency.

Table 5.1 contains the dynamic bytecode instruction frequencies measured by running our benchmarks with an instrumented interpreter and collecting/aggregating the instruction frequencies whilst running the benchmarks. Summing this data over all of our benchmarks and taking the top-most 15 instructions and their corresponding frequencies gives this table. First, we see that optimized interpreter executes slightly more instructions, 5,980 interpreter instructions, or about 0.0008% more. Because of detailed comparison of the instruction frequency distribution among the computationally most relevant function of the corresponding benchmarks give identical instruction frequencies, we assume that this difference is due to interpreter internals, such as loading libraries from different directories—given that the difference is numerically insignificant, however, this has only negligible influence on the following quantitative evaluation.

First of all, Table 5.1 shows that instructions loading operands onto the stack constitutes the biggest part of all executed instructions:

- `LOAD_FAST` instructions constitute 28.50%,
- `LOAD_FAST` and `LOAD_CONST` instructions make up 41.98% of all executed instructions,
- `LOAD_FAST`, `LOAD_CONST`, and `STORE_FAST` instructions total over half of all instructions, viz., 51.86%.

Hence, the top 3 executed instructions make up more than half of all executed instructions in the standard Python 3.1 interpreter. Therefore, some of our

optimizations, such as partial stack frame caching (cf. Section 3.5), and data object inlining (cf. Section 3.1.2) are particularly beneficial.

First, lets consider the effectiveness of data object inlining:

- `LOAD_CONST`: 98.78% of constant operand objects can be inlined by `INCA_LOAD_CONST`, and `INCA_LOAD_CONST_NORC`,
- `LOAD_GLOBAL`: 76.51% of global operand objects can be inlined by `INCA_LOAD_GLOBAL`, and `INCA_LOAD_GLOBAL_NORC`.

Next, we evaluate the effectiveness of partial stack frame caching. A glance at Table 5.1 already indicates that using our heuristic works reasonably well, since the dynamic bytecode frequencies of the optimized derivative instructions are in decreasing order of their scores, i.e., from `LOAD_FAST_A` to `LOAD_FAST_D`:

- 28.50% of `LOAD_FAST` instructions can be promoted to `LOAD_FAST_A` alone,
- 75.65% of `LOAD_FAST` instructions can be optimized by using partial stack frame caching instructions,
- 64.67% of `STORE_FAST` instructions can be promoted to using partial stack frame caching instructions.

Finally, we analyze the effect of reference count quickening (cf. Section 3.4) on the load instructions:

- 75.94% of `LOAD_CONST` instruction's reference count operations are redundant,
- 76.51% of `LOAD_GLOBAL` instruction's reference count operations are redundant,
- 85.21% of all `LOAD_FAST` instruction's reference count operations are redundant.

Interestingly, none of the top most-frequent `STORE_FAST` instruction occurrences are optimized by reference count quickening, which suggests that most `STORE_FAST` occurrences store data objects with an implicit reference count operation, as is the result of any arithmetic operation instruction.

In consequence, this evaluation shows that using these optimization techniques apply to frequently occurring cases that support evidence of the applicability of these techniques, since *more than three quarters* of load instructions can be optimized.

5.2.2 Analysis of Local Variables

This section presents detailed data on the call frequency—more specifically function stack-frames, where we are interested in the number of local variables the stack frames. This helps us to examine the exact number of cache slots to use in combination with partial stack frame caching (cf. Section 3.5).

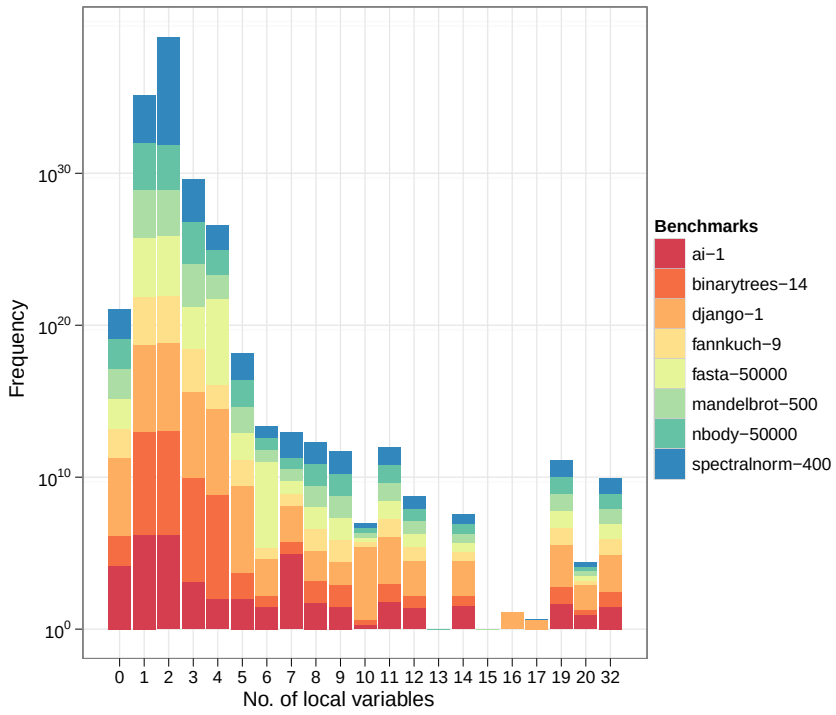


Figure 5.1: Distribution of the number of local variables per stack frame.

Figure 5.1 shows the frequency of processed functions broken down by the number of local variables allocated in each stack frame. Moreover, Figure 5.1 displays the gathered data for all benchmarks, which allows us to make informed decisions concerning several distinct applications.

Number of Local Variables	Coverage	Number of Local Variables	Coverage
0	00.3322%	6	99.6401%
1	18.9031%	7	99.8416%
2	66.4442%	8	99.8423%
3	81.5752%	9	99.8428%
4	**97.5539%**	10	99.9932%
5	98.7543%		

Table 5.2: Total coverage of calls covered per number of local variables.

Table 5.2 displays the percentile coverage of calls. This shows us which number of local variables is most frequent. When we compare these results with Figure 5.1, we can verify that indeed, stack frames with one or two local variables dominate the measured frequencies. Furthermore, we have marked the entry having four local variables in Table 5.2: about 97.5% of calls have at most four local variables. Therefore, we decide to use four cache slots for partial stack frame caching Section 3.5).

5.2.3 Analysis of Function Calls

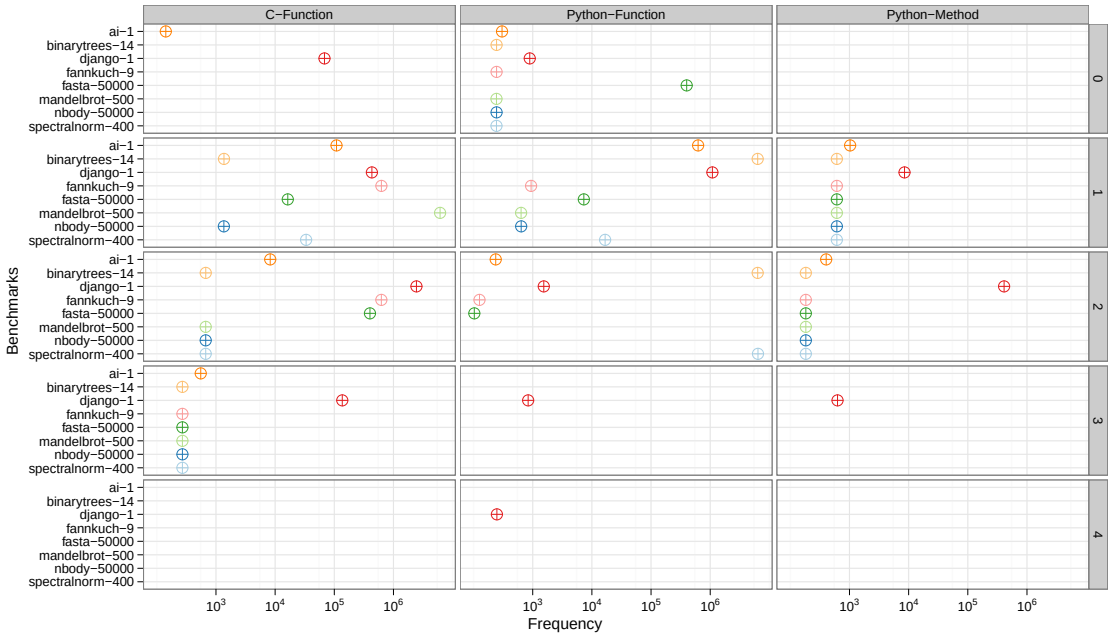


Figure 5.2: Frequencies of call types grouped by number of arguments and call targets.

Figure 5.2 shows the results we obtained by running several benchmarks and collecting several function call data. We eliminated 69 entries amounting to a total of 1,654 function call instructions—for argument tuples of size 5, 6, and 7—out of 32,723,825 totally collected `CALL_FUNCTION` instructions. This was necessary to eliminate the vertical grouping for the corresponding argument tuple sizes, i.e., the figure only shows groups for argument tuple sizes from 0, which corresponds to a function call without any argument, up to an argument tuple size of 4. Horizontally, the data is grouped by the call target of the `CALL_FUNCTION` instruction, i.e., grouped by whether the call target is either a C-Function, a Python-Function, or a Python-Method.

For each `CALL_FUNCTION` grouped by call target, Figure 5.2 shows that either one or two arguments are among the most frequently occurring `CALL_FUNCTION` instructions. In general, the common trend is that from that peak, the frequency

of `CALL_FUNCTION` instructions with a larger argument tuple decreases, i.e., is less likely overall. Note that this corresponds very well with the real-world `django` benchmark (cf. the red cross in Figure 5.2), too. For example, calls to C functions have a peak at argument tuple size two, calls to Python functions a peak at argument tuple size one, and the most frequent calls to Python methods within the `django` benchmark have two arguments. Hence, this data confirms our choices for the selection of which argument sizes and call types to provide for our optimized inline cached call instructions (cf. Table 3.3, and Section 3.3.5.)

5.3 Analysis of Reference Count Operations

Benchmark	Reference Count Operations				Instruction Frequency
	Standard		Optimized		
	Increment	Decrement	Increment	Decrement	
ai-1	35,084,658	38,820,194	27,329,155	30,609,845	26,894,583
binarytrees-14	184,145,592	203,145,824	127,161,852	155,681,200	203,062,096
django-1	60,374,548	74,326,819	50,809,784	62,629,187	41,027,619
fannkuch-9	100,000,143	107,032,592	72,121,596	77,906,679	77,004,917
fasta-50000	11,045,307	13,999,691	7,015,288	9,161,436	11,053,675
mandelbrot-500	53,052,259	72,879,417	13,667,179	33,494,333	93,122,405
nbody-50000	62,868,435	76,534,735	26,517,643	40,183,938	68,976,753
spectralnorm-400	149,402,835	211,900,227	78,936,240	147,817,776	205,234,415

Table 5.3: Number of reference count operations per benchmark.

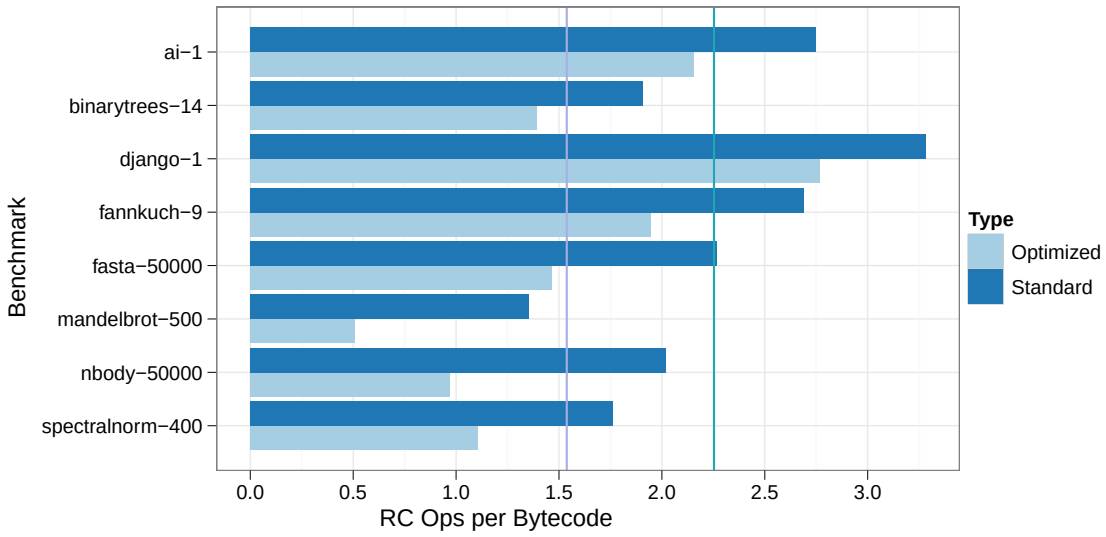


Figure 5.3: Reference count operations per bytecode.

Figure 5.3 presents a figure from the raw data of Table A.28. The left vertical line corresponds to the average number of reference count operations per bytecode when using reference count quickening (cf. Section 3.4), whereas the vertical line to the right corresponds to the number of reference count operations in the standard interpreter. We can see that using reference count quickening reduces the reference count operations per bytecode from about 2.25 to 1.5—a reduction by a third or 33.3%.

5.4 Performance Evaluation

5.4.1 Detailed Speedup Factors

Tables 5.4, 5.5 and 5.6 present our calculated speedup factors per benchmark. Our modified version of the `nanobench` program calculates the average time each interpreter needs to process a benchmark by summing up all run-times recorded and dividing this aggregate by the number of runs. In this specific averaging process, we do not care about the instances of the arguments supplied to each benchmark. For example, if we run the `binarytrees` benchmark 50 times with different arguments of 10, 12, and 14, our average value sums up all recorded times and divides them by 150. The rationale for this is that the interpreters perform well on specific benchmarks and not on specific argument instances. We calculate the speedup relative to the timing results of the vanilla Python 3.1 interpreter. We provide detailed run-time data for each benchmark and interpreter configuration in the appendix (cf. Appendix B.) Because of the Intel Atom N270 having only a 32bit instruction size, we do not report data for each optimization technique separately, since some of them are not available, such as data object reference inlining. Therefore, we only report interpreter instruction scheduling data for the Intel Atom N270.

Interpreter	Speedup factors per benchmark						
	`binary-trees`	`fannkuch`	`fasta`	`mandel-brot`	`nbody`	`spectral-norm`	Overall
Vanilla	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
+ Threaded Code	1.4525	1.4544	1.2902	1.5686	1.3173	1.5867	1.4330
+ New Instr. Format	1.3389	1.3712	1.3675	1.5487	1.2864	1.4190	1.3698
+ PSFC[1]	1.3846	1.3940	1.3913	1.6239	1.2498	1.5931	1.4083
+ Load Caching	1.4739	1.3927	1.3389	1.5790	1.2734	1.5509	1.4188
+ Inline Caching	1.8500	1.5508	1.4989	2.0342	1.5367	2.2518	1.7646
+ RCQ[2]	1.9257	1.6509	1.5446	2.0040	1.6165	2.2538	1.8213
IIS[3]/binarytrees	1.9891	1.6925	1.5148	2.1084	1.7301	2.4064	1.9028
IIS/fannkuch	1.9748	1.6964	1.5861	2.1072	1.7602	2.4064	1.9203
IIS/fasta	1.9746	1.6824	1.7161	2.1315	1.7565	2.4146	1.9382
IIS/mandelbrot	1.9744	1.7264	1.6077	2.1847	1.7277	2.3796	1.9213
IIS/nbody	1.9713	1.7211	1.6081	2.0995	1.7557	2.3710	1.9186
IIS/spectralnorm	1.9780	1.5866	1.4600	2.1255	1.6941	**2.4176**	1.8714

Table 5.4: Speedup factors per benchmark for all interpreter configurations on the Intel Nehalem i7-920.

Interpreter	Speedup factors per benchmark						
	`binary-trees`	`fannkuch`	`fasta`	`mandel-brot`	`nbody`	`spectral-norm`	Overall
Vanilla	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
+ Threaded Code	1.1469	1.1664	1.0509	1.1020	1.3885	1.1271	1.1854
+ New Instr. Format	1.0746	1.1856	1.0662	1.1514	1.4230	1.1447	1.1900
+ PSFC	1.0356	1.0843	0.9829	1.0520	1.2497	1.0774	1.0971
+ Load Caching	1.1690	1.0651	1.0172	1.0976	1.2949	1.0829	1.1469
+ Inline Caching	1.4173	1.2506	1.0619	1.1283	1.5689	1.4194	1.3445
+ RCQ	1.3818	1.3000	1.1197	1.1398	1.6330	1.3319	1.3513
IIS/binarytrees	1.4852	1.2951	1.1190	1.0778	1.5973	1.5405	1.3836
IIS/fannkuch	1.4394	1.3357	1.1816	1.1650	**1.7000**	1.4829	1.4199
IIS/fasta	1.4465	1.2561	1.1726	1.1499	1.5715	1.5885	1.3977
IIS/mandelbrot	1.4878	1.2812	1.0930	1.2842	1.6489	1.5731	1.4371
IIS/nbody	1.3747	1.3168	1.0986	1.1623	1.5882	1.5084	1.3728
IIS/spectralnorm	1.4343	1.2403	1.1508	1.2681	1.5660	1.5199	1.4022

Table 5.5: Speedup factors per benchmark for all interpreter configurations on the PowerPC 970.

We report our highest speedup factor by 2.4176 for the `spectralnorm` benchmark on the Intel Nehalem i7-920 (cf. Table 5.4, bold digits.) In general, the performance improvements of our purely interpretative optimization techniques are substantial: two benchmarks more than double their performance, one almost doubles the performance and the remaining three at least improve performance by at least 50%.

The optimization potential on the PowerPC 970 CPU is not on par with the Intel Nehalem i7-920. Nevertheless, we report a maximum overall speedup factor by 1.70 for the `nbody` benchmark on that system. Including our interpreter instruction scheduling technique, we report an average speedup factor by 1.4371 or about 44% using the `mandelbrot`'s schedule, which performs very well across several benchmarks.

The following sub-sections present a careful and detailed analysis of the figures within the tables, first we are going to analyze the impact of each optimization technique and, second we are going to discuss the effects of interpreter instruction scheduling.

5.4.2 Results per Optimization Technique

Threaded Code Instruction Dispatch Optimization

Contrary to the result of Vitale and Abdelrahman [VA04], we have not found any slowdowns by enabling the threaded code instruction dispatch optimization in the Python interpreter. However, we find that its performance is significantly below the usually reported figures of 2.02 (cf. Table 2.1.) While it performs very

[1]Partial Stack Frame Caching.
[2]Reference Count Quickening.
[3]Interpreter Instruction Scheduling.

Interpreter	Speedup factors per benchmark						
	`binary-trees`	`fannkuch`	`fasta`	`mandel-brot`	`nbody`	`spectral-norm`	Overall
... + RCQ	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
IIS/binarytrees	1.1131	1.0090	0.9998	0.9688	1.0073	1.0323	1.0257
IIS/fannkuch	1.0119	1.1158	1.0182	0.9573	1.0478	1.0289	1.0241
IIS/fasta	1.0428	1.0098	1.0923	0.9238	1.0633	0.9920	1.0237
IIS/mandelbrot	1.0398	1.0521	1.0038	1.0736	1.0266	0.9817	1.0263
IIS/nbody	1.0173	1.0473	1.0011	0.9215	1.0902	0.9971	1.0129
IIS/spectralnorm	1.0176	1.0580	1.0188	0.9630	1.0361	**1.1344**	1.0354

Table 5.6: IIS Speedup factors per benchmark on the Intel Atom N270.

well on the Intel i7-920, its performance on the PowerPC 970 is substantially lower. On the Intel i7-920, the performance is almost 21% higher ($\frac{1.4330}{1.1854}$.)

New Instruction Format

Switching to our new instruction format of Section 3.1 reduces performance on the Intel i7-920 and improves performance on the PowerPC 970 (there is but one exception on both systems.) Neither increase ($\approx 0.4\%$, $\frac{1.1900}{1.1854}$) nor decrease ($\approx 5\%$, $\frac{1.4330}{1.3698}$) in performance are particularly noteworthy, yet some of our follow-up optimization techniques require the increased bytecode size and need more than 255 instructions representable in the standard byte-sized bytecode instruction format.

Partial Stack Frame Caching

On our Intel i7-920 system, partial stack frame caching (cf. Section 3.5) shows a varying performance potential. While we mostly report improved performance over the previous results with the new instruction format and the threaded code version, the performance is only sometimes better than the original threaded code version alone. To make matters worse, the performance on the PowerPC 970 is significantly below what we expected from our figures on the Intel i7-920 system. In fact, for the `fasta` benchmark we found that partial stack frame caching reduces performance. From the dynamic bytecode frequencies for the `fasta` benchmark, we note that this specific benchmark would benefit a lot from `LOAD_DEREF` instructions via partial stack frame caching, too.

Load Caching, Data Object Reference Inlining

Enabling load caching, i.e., the inlining of data object references for the `LOAD_GLOBAL`and `LOAD_CONST` instructions (cf. Section 3.1.2) gives similar results as what we have just discussed for partial stack frame caching. Again, our results show that the effective optimization potential is highly dependent on the actual benchmark.

Inline Caching

Using our quickening based inline caching technique from Section 3.3 gives the biggest speedup factors across both processor architectures. For the Intel i7-920, we report a maximum speedup by about 42% ($\approx \frac{2.2518}{1.5867}$) when compared to the threaded code only version. For the PowerPC 970, we report a maximum speedup by about 26% ($\approx \frac{1.4194}{1.1271}$) when compared to the threaded code only version. We report both maximum speedup factors for the `spectralnorm` benchmark. Over all benchmarks, we report an increase of performance by about 23% ($\approx \frac{1.7646}{1.4330}$) on the Intel Nehalem architecture and an increase by about 13% ($\approx \frac{1.3445}{1.1854}$) on the PowerPC 970, when compared to the performance of the corresponding threaded code only interpreters.

Reference Count Quickening

Adding reference count quickening (cf. Section 3.4) to the previous inline caching based interpreter increases performance by about 3% ($\approx \frac{1.8213}{1.7646}$) on the Intel i7-920, and only moderately by half a percent ($\approx \frac{1.3513}{1.3445}$) on the PowerPC 970. The corresponding maximum performance improvements occur on the `fannkuch` benchmark for the Intel Nehalem architecture ($6.5\% \approx \frac{1.6509}{1.5508}$), and on the `spectralnorm` benchmark for the PowerPC architecture ($6.6\% \approx \frac{1.4194}{1.3319}$.)

Following Figure 3.28 of Section 3.7 in combination with the encouraging results of interpreter instruction scheduling (see next sub-section, Section 5.4.3), we assume that the performance of reference count quickening suffers from a significant impact of instruction cache penalties caused by the numerous new instruction derivatives.

5.4.3 Interpreter Instruction Scheduling

In addition to the speedup factors broken down by their respective optimization technique and in-depth discussion and evaluation of the performance benefits, Tables 5.4, 5.5, and 5.6 contain data for interpreter instruction scheduling with schedules for each benchmark. Again, we report this aggregated data based on our detailed results from the appendix (cf. Appendix B.)

For the Intel Nehalem i7-920 and the PowerPC 970, this data evidently states that there is a non-causal relationship between the actual instruction schedules and the benchmark programs themselves. For example, for the `fannkuch` benchmark running on the Intel i7-920, the schedule computed by the kernel of the `mandelbrot` benchmark performs best. Similarly, on the PowerPC 970 the interpreter with the `fannkuch` schedule performs best for the `nbody` benchmark. On the other hand, however, on the Intel Atom N270 Table 5.6 displays a causal relationship between the computed instruction schedule and the corresponding benchmarks.

Another interesting effect of interpreter instruction scheduling is that there seems to be a set of instruction sequences which are particularly well covered by some computed schedules. The first evidence is that for both CPU architectures, the overall scores for all interpreters with different instruction schedules perform better than the versions without instruction scheduling. Supplementary evidence is that there are some schedules which perform particularly well over all benchmarks leading to a maximum combined speedup. For the Intel Nehalem i7-920,

an interpreter with the `fasta` instruction schedule achieves the highest overall speedup by a factor of 1.9382, which corresponds to an improvement of about 6% ($\approx \frac{1.9382}{1.8213}$) over the reference count quickening results, and an improvement of about 10% ($\approx \frac{1.9382}{1.7646}$) over the interpreter with inline caching. On the Intel Atom N270, we measured the biggest overall speedup by 3.54% by using the `spectralnorm` instruction schedule.

On the PowerPC 970, the interpreter with the `mandelbrot` instruction schedule performs best from an overall perspective. When compared to the reference count quickening interpreter, we report a performance improvement by about 6% ($\approx \frac{1.4371}{1.3513}$). When compared to the inline caching interpreter, we report a performance improvement by about 7% ($\approx \frac{1.4371}{1.3445}$.)

When regarding the maximum potential of interpreter instruction scheduling, we examine the `spectralnorm` benchmark on the PowerPC 970: Using the `fasta` instruction schedule, the performance improves by substantial 19% ($\approx \frac{1.5885}{1.3319}$) when compared to the reference count quickening interpreter. This conclusively shows that the instruction cache miss penalties caused by the considerable instruction set extension can be successfully overcome using interpreter instruction scheduling. With an optimal interpreter instruction schedule, reference count quickening can full unfold its additional potential: When compared to the inline caching only interpreter, the performance improvement still makes up about 12% ($\approx \frac{1.5885}{1.4194}$.)

In conclusion, the evaluation of interpreter instruction scheduling demonstrates the following:

- *Applicability* of interpreter instruction scheduling: Our data indicates that some instruction schedules perform better than others for different input programs. Therefore, it is merely an engineering problem to provide interpreters that offer premium performance for given application profiles, such as customized Python interpreters for the django web application framework.

- *Maximum performance* of interpreter instruction scheduling: When maximum performance for any interpreter is necessary, profiling and experimenting of the application in question can yield substantial benefits and is therefore highly recommended.

Chapter 6

Conclusions

In conclusion, this thesis presents our efforts in optimizing interpreters. More specifically, we demonstrate the potential of purely interpretative optimization techniques, which are particularly well suited to optimizing high abstraction-level interpreters, because these techniques target operation implementation instead of instruction dispatch. We report significant overall speedups of up to 2.4176.

In addition to the competitive speedup potential, the techniques themselves are simple in nature, which results in rapid implementation times with only little effort. This is in stark contrast to the efforts necessary for implementing a dynamic compilation sub-system and therefore presents an important intermediate step for optimizing virtual machines *before* having to commit valuable resources to implementing a just-in-time compiler.

6.1 Summary of Contributions

- We discuss the historical perspective of using purely interpretative inline caching based on interleaving pointers. Furthermore, we discuss how this technique compares to the well-known approach of using hash-table based look-up caches.
- We introduce a very efficient inline caching technique based on quickening, which achieves substantial speedups. In addition to the application to inline cache call instructions, we present their use in optimization of other instructions, such as iteration and comparison instructions.
- We present partial stack-frame caching to reduce the overhead in the most frequently executed instructions of stack based interpreters: load and store instructions. In combination with the elimination of redundant reference count operations and proper register allocation, this allows us to minimize the operation implementation of a load instruction to just one assembly instruction.
- We show our approach to eliminate redundant reference count operations in immediate reference counting, effectively providing an alternative to deferred reference counting [DB76]. Depending on the benchmark, this technique is able to eliminate up to two thirds of increment and up to

half of all decrement reference count operations. These rates could be significantly higher if we were to add dedicated operation implementation functions without implicit reference count operations.

- We show how to use interpreter instruction scheduling to improve the instruction cache utilization of interpreters. This results in additional speedup factors of up to 1.14 on a modern Intel i7 Nehalem, and of average speedup of about 10% on Intel Atom architectures. While this technique in its static implementation might seem unpractical for interpreters at first, there is a need for either a) high performance server environments with large scalability requirements can easily have multiple different interpreter binaries with distinctive cache utilization patterns, and b) mobile devices which would benefit not only by improved performance but actually from reduced energy consumption. Here, too, customized binaries can be easily used.

6.2 Future Work

We can categorize future work into two classes: i) demonstrate applicability, and practicability, and ii) new research. We hope to demonstrate the applicability of the optimizations presented herein by implementing them in other high abstraction-level virtual machines, such as the JavaScript, Perl, and Ruby interpreters. A very interesting project would be to add these optimizations to functional and/or logical programming language implementations, too.

Among the low hanging fruit for new research and development is to further enhance the performance of our purely interpretative inline caching technique by inlining the function bodies of the functions implementing the actual operations, e.g., inlining the `float_add` function body pointed to by the corresponding type structure `PyFloat_Type.tp_as_number->nb_add`. Theoretically, any compiler could do this already, however, `gcc` 4.4.3 does not do it, because the inlining is restricted to module boundaries, i.e., the `PyFloat_Type` type and implementation live in their own C module and cannot be inlined by `gcc` 4.4.3 . We estimate the performance impact of inlining for important operators as well as frequently used iterators to be substantial.

Similar to the recent result of Williams, McCandless, and Gregg [WMG10], one could use our optimization techniques to dynamically generate optimized instruction derivatives using a bytecode rewrite toolkit for virtual machine implementations hosted on an existing virtual machine infrastructure with a dynamic compilation sub-system. Consequently, we can significantly optimize hosted implementations of Python (Jython), and Ruby (JRuby).

In our earlier discussion of overheads in operation implementation we identify that many high abstraction-level virtual machines have significant overhead due to any combination of the following three features:

- dynamic typing,
- reference counting,
- (un-)boxing of objects.

Subsequently, we presented techniques to optimize the first two of the three inefficiencies. However, we did not present an optimization to attack (un-)boxing of objects. A traditional optimization approach is to use a tagged pointer representation to combine multiple data object references into just one pointer. Now, any operation implementation has to check the type tag and choose the actual operation. Besides limiting the address range available to the implementer for regular data object reference pointers, this involves relatively expensive tag checks.

Our idea to remove the frequently used tagged data representation is to combine multiple techniques presented in this thesis. First, we create a simple abstract interpreter similar to the one to find sequences of redundant reference count operations, but this new simple abstract interpreter finds sequences of operations that operate on the same data objects. Next, we provide additional optimized derivatives that operate on primitive values, instead of boxed objects. Then, we create optimized derivatives of load and store instructions that unbox and box objects correspondingly. Finally, we combine all previous steps and quicken general instruction sequences to sequences of using primitive data types instead. Note that this subsumes inline caching and reference counting, too.

6.3 Interpreter Optimization Recommendations

1. Use an interpreter generator, such as `vmgen` [EGKP02]. To generate multiple derivatives, one could use a generator that generates `vmgen` source files. This simplifies experimentation and development considerably. In addition, one could use the configuration infrastructure of a project to customize interpreter optimizations to the actual user's needs.

2. Use a regular instruction format, even at the expense of additional memory. Ensure that instruction en- and decoding is as efficient as possible (cf. Section 3.1).

3. Use threaded code optimization: Is highly likely to give speedups and is an interpreter implementation best-practice.

4. Quantitative analysis to determine concrete value instance for optimization variables, such as the number of partial stack frame caching slots.

5. Use partial-stack-frame caching to speed-up loads—this is independent of abstraction-level and applies to almost all interpreters using a stack architecture (cf. Section 3.5).

6. Determine the abstraction level of the interpreter and choose optimization techniques correspondingly.
   If you have a *high* abstraction-level virtual machine:

   (a) Use quickening based operation unfolding for expensive instruction-operand-dependent implementations.

   (b) Use purely interpretative quickening based inline caching for a dynamically-typed programming language/interpreter (cf. Section 3.3).

   (c) Eliminate overhead of reference counting using our technique from Section 3.4.

   If you have a *low* abstraction-level virtual machine:

   (a) Use superinstructions [EG03a].

   (b) Use replication [EG03a]

7. Use interpreter instruction scheduling to optimize the interpreter execution of a specific program on the target hardware, particularly worthwhile on smaller architectures, such as 32bit x86, and PowerPC RISC architectures.

Bibliography

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003. Cited on page 12.

[Bad82] Scott B. Baden. High performance storage reclamation in an object-based memory system. Technical report, Berkeley, CA, USA, 1982. Cited on page 41.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977. Cited on pages 71 and 72.

[Bay10] Michael Bayer. Mako. `http://www.makotemplates.org`, May 2010. Cited on page 64.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, Lecture Notes in Computer Science, pages 18–25. Springer, 2009. Cited on pages 2 and 28.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. Cited on page 63.

[Bra10] Gilad Bracha. Private communication. June 2010. Cited on page 32.

[Bru09] Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09)*, volume 253(5) of *Electronic Notes in Theoretical Computer Science*, pages 3–14, Amsterdam, The Netherlands, December 2009. Elsevier. Cited on pages iv, x, and 7.

[Bru10a] Stefan Brunthaler. Efficient inline caching without dynamic translation. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, pages 2155–2156, New York, NY, USA, March 2010. ACM. Cited on pages iv, 26, and 33.

[Bru10b] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages, Reno, Nevada, US, October 18, 2010 (DLS '10)*, New York, NY, USA, 2010. ACM Press. To appear. Cited on pages iv, 26, 33, 47, and 63.

[Bru10c] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25, 2010 (ECOOP '10)*, volume 6183/2010 of *Lecture Notes in Computer Science*, pages 429–451. Springer, 2010. Cited on pages iv, 26, and 33.

[Bru11] Stefan Brunthaler. Interpreter instruction scheduling. In *Proceedings of the 14th International Conference on Compiler Construction, Saarbrücken, Germany, March 26-April 3rd, 2010 (CC '11)*, Lecture Notes in Computer Science. Springer, 2011. To appear. Cited on pages iv and 53.

[CEG05] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizations for a java interpreter using instruction set enhancement. Technical Report 61, Department of Computer Science, University of Dublin. Trinity College, September 2005. Cited on page 71.

[Cha92] Craig David Chambers. *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages.* PhD thesis, Stanford, CA, USA, 1992. Cited on page 12.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960. Cited on pages 9, 10, 40, and 71.

[CPL82] Thomas J. Conroy and Eduardo Pelegri-Llopart. *An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations*, chapter 13, pages 239–247. In Krasner [Kra84], 1982. Cited on pages 29 and 70.

[CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2004. Cited on page 53.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976. Cited on pages 10, 40, 41, 71, 72, and 86.

[dja11] Django. `http://www.djangoproject.com/`, January 2011. Cited on page 75.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the SIGPLAN '84 Symposium on Principles of Programming Languages (POPL '84)*, pages 297–302, New York, NY, USA, 1984. ACM. Cited on pages 11, 26, 27, 28, and 32.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag. Cited on page 8.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 278–288, New York, NY, USA, 2003. ACM. Cited on pages 7, 74, and 89.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003. Cited on pages 6 and 7.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Software Practice & Experience*, 32:265–294, March 2002. Cited on page 89.

[Ful] Brent Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`. Cited on pages 56, 58, and 75.

[GES+09] 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 *Proceedings of the SIGPLAN '09 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 465–478, New York, NY, USA, 2009. ACM. Cited on pages 1 and 12.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. Cited on pages 9, 10, and 29.

[Gri96] Robert Griesemer. Strongtalk source code. `http://code.google.com/p/strongtalk/source/browse/trunk/vm/interpreter/interpreter.cpp#1931`, June 1996. Lines 1931ff, checked October 2010. Cited on page 32.

[HCU91] Urs Hölzle, Craig Chambers, and David M. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991 (ECOOP '91)*, volume 512/1991 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991. Cited on pages 26, 28, and 70.

[HHD07] Michael Haupt, Robert Hirschfeld, and Marcus Denker. Type feedback for bytecode interpreters. Position Paper. (ICOOOLPS '07). In *Proceedings of the 2nd Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '07)*, Lecture Notes in Computer Science. Springer, 2007. Cited on page 70.

[Höl94] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford

University, Stanford, CA, USA, 1994. Cited on pages 12, 32, 36, and 70.

[Höl09] Urs Hölzle. Private communication. September 2009. Cited on page 32.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 326–336, 1994. Cited on page 26.

[Int08] Intel. Intel Turbo Boost Technology in Intel Core microarchitecture (Nehalem) based processors. Online, November 2008. Cited on page 75.

[JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. Cited on page 11.

[Joi06] Pramod G. Joisha. Compiler optimizations for nondeferred reference counting garbage collection. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*, pages 150–161, New York, NY, USA, 2006. ACM. Cited on page 72.

[Joi08] Pramod G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, pages 131–140, New York, NY, USA, March 2008. ACM. Cited on page 72.

[Jon07] Richard Jones. Dynamic memory management: Challenges for today and tomorrow. In *Proceedings of the International Lisp Conference, Cambridge, UK, March 31-April 5, 2007*, pages 115–124, 2007. Cited on page 11.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*, pages 99–105, New York, NY, USA, 1990. ACM. Cited on pages 29 and 70.

[Kra84] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983, reprinted with corrections 1984. Cited on pages 29, 70, 91, and 95.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008. Cited on page 12.

[LC08] Chun-Chieh Lin and Chuen-Liang Chen. Code arrangement of embedded java virtual machine for NAND flash memory. In Per Stenström, Michel Dubois, Manolis Katevenis, Rajiv Gupta, and

Theo Ungerer, editors, *Proceedings of the Third High Performance Embedded Architectures and Compilers International Conference, Göteborg, Sweden, January 27-29, 2008 (HiPEAC '08)*, volume 4917 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2008. Cited on page 73.

[Ler03] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003. Cited on pages 42 and 73.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Boston, MA, USA, first edition, 1996. Cited on pages 33, 71, and 72.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. Cited on pages 4 and 40.

[Mir87] Eliot Miranda. Brouhaha–a portable smalltalk interpreter. In *Proceedings of the SIGPLAN '87 International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87)*, pages 354–365, New York, NY, USA, 1987. ACM. Cited on page 10.

[Mor98] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. Cited on page 53.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. Cited on page 53.

[ope10] Open Research Compiler. `http://ipf-orc.sourceforge.net/`, October 2010. Cited on page 73.

[Pal10] Mike Pall. Better use an interpreter. `http://lambda-the-ultimate.org/node/3851#comment-57646`, March 2010. Exact time stamp: 21:05, March 9, 2010. Cited on page 10.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, 1990. Cited on page 73.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 291–300, New York, NY, USA, 1998. ACM. Cited on pages 7 and 8.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programmming Languages and Systems (TOPLAS)*, 21:895–913, September 1999. Cited on page 49.

[RLPN+99] Alex Ramírez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceedings of the 13th International Conference on Supercomputing, Rhodes, Greece, June 20-25, 1999 (ICS '99)*, pages 119–126, New York, NY, USA, 1999. ACM. Referenced by gcc/bb-reorder.c. Cited on page 63.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159. ACM Press, 1996. Cited on page 6.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008. Cited on pages 7 and 47.

[SPS09] Richard M. Stallmann, Roland H. Pesch, and Stan Shabs. *Debugging with GDB: The GNU source-level debugger*. Free Software Foundation, 9th edition, 2009. Cited on page 65.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167, New York, NY, USA, 1984. ACM. Cited on page 10.

[Ung86] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, EECS Department, University of California, Berkeley, February 1986. UCB/CSD-86-287. Cited on pages viii, 40, and 41.

[unl10] Unladen Swallow. `http://code.google.com/p/unladen-swallow/`, August 2010. Cited on page 75.

[UP82] David M. Ungar and David A. Patterson. *Smalltalk-80: Bits of History, Words of Advice*, chapter 11, Berkeley Smalltalk: Who Knows Where the Time Goes?, pages 189–206. In Krasner [Kra84], September 1982. Cited on page 72.

[US07] David Ungar and Randall B. Smith. Self. In *Proceedings of the 3rd ACM SIGPLAN conference on History of programming languages (HOPL III)*, pages 9–1–9–50, New York, NY, USA, 2007. ACM. Cited on page 9.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)*, pages 42–50, New York, NY, USA, 2004. ACM. Cited on pages 7, 8, and 82.

[vL10] Martin von Löwis. Porting django to 3k. `http://wiki.python.org/moin/PortingDjangoTo3k`, April 2010. Cited on page 75.

[WB82] Allen Wirfs-Brock. *Smalltalk-80: Bits of History, Words of Advice*, chapter 4, Design Decisions for Smalltalk-80 Implementors, pages 41–56. In Krasner [Kra84], 1982. Cited on pages 9 and 29.

[Whe10] David Wheeler. sloccount. `http://www.dwheeler.com/sloccount/`, May 2010. Cited on page 68.

[WMG10] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM SIGMICRO/SIGPLAN International Symposium on Code Generation and Optimization (CGO '10)*, pages 278–287, April 2010. Cited on page 87.

[YWF09] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS '09)*, pages 79–88, New York, NY, USA, 2009. ACM. Cited on pages 2 and 28.

[ZA05] Peng Zhao and José Nelson Amaral. Feedback-directed switch-case statement optimization. In *Proceedings of the International Conference on Parallel Programming Workshops, Oslo, Norway, June 14-17 2005 (ICPP '05 Workshops)*, pages 295–302. IEEE, August 2005. Cited on page 73.

Appendix A

Detailed Benchmark Evaluation

A.1 Binarytrees

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	57,068,356	INCA_LOAD_CONST_NORC	22,145,774
2	LOAD_CONST	28,542,718	LOAD_FAST_B_NORC	15,821,518
3	STORE_FAST	25,390,930	LOAD_FAST_C_NORC	15,821,512
4	POP_JUMP_IF_FALSE	12,698,922	LOAD_FAST_A	12,713,996
5	LOAD_GLOBAL	12,696,573	POP_JUMP_IF_FALSE	12,698,944
6	CALL_FUNCTION	12,696,408	RETURN_VALUE	12,693,728
7	COMPARE_OP	12,695,344	INCA_LOAD_GLOBAL_NORC	9,519,126
8	RETURN_VALUE	12,693,728	STORE_FAST_C	9,519,122
9	BUILD_TUPLE	6,346,416	STORE_FAST_B	9,497,282
10	UNPACK_SEQUENCE	6,346,339	INCA_LOAD_CONST	6,389,787
11	BINARY_SUBTRACT	6,302,429	LOAD_FAST_B	6,367,924
12	BINARY_ADD	3,175,595	STORE_FAST_D	6,346,084
13	BINARY_MULTIPLY	3,151,274	INCA_BUILD_TUPLE_THREE	6,346,084
14	INPLACE_SUBTRACT	3,151,198	COMPARE_OP_NORC	6,346,079
15	FOR_ITER	24,569	INCA_UNPACK_TUPLE_THREE	6,346,078

Table A.1: Comparative dynamic instruction frequency for the `binarytrees` benchmark (Argument: 14).

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	6,346,078	1	LOAD_FAST_B_NORC	16	3,151,198	16	INCA_LOAD_CONST_NORC
2	6,346,078	2	INCA_LOAD_CONST_NORC	17	3,151,198	17	INCA_LONG_SUBTRACT_NORC
3	6,346,078	3	INCA_CMP_LONG_NORC	18	3,151,198	18	LOAD_FAST_B
4	6,346,078	4	POP_JUMP_IF_FALSE	19	3,151,198	19	FAST_PYFUN_TWO
5	3,151,198	5	INCA_LOAD_CONST_NORC	20	3,151,198	20	INCA_LOAD_GLOBAL_NORC
6	3,151,198	6	LOAD_FAST_A_NORC	21	3,151,198	21	LOAD_FAST_C_NORC
7	3,151,198	7	INCA_LONG_MULTIPLY_NORC	22	3,151,198	22	LOAD_FAST_B_NORC
8	3,151,198	8	STORE_FAST_C	23	3,151,198	23	FAST_PYFUN_TWO_NORC
9	3,151,198	9	LOAD_FAST_B_NORC	24	3,151,198	24	INCA_BUILD_TUPLE_THREE
10	3,151,198	10	INCA_LOAD_CONST_NORC	25	3,151,198	25	RETURN_VALUE
11	3,151,198	11	INCA_LONG_SUBTRACT_NORC	26	3,194,880	26	LOAD_FAST_A
12	3,151,198	12	STORE_FAST_B	27	3,194,880	27	INCA_LOAD_CONST
13	3,151,198	13	LOAD_FAST_A	28	3,194,880	28	INCA_LOAD_CONST
14	3,151,198	14	INCA_LOAD_GLOBAL	29	3,194,880	29	INCA_BUILD_TUPLE_THREE
15	3,151,198	15	LOAD_FAST_C_NORC	30	3,194,880	30	RETURN_VALUE

Table A.2: Instruction trace and frequency for `make_tree` function of `binarytrees` benchmark.

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	6,346,078	1	LOAD_FAST_A	12	3,151,198	12	LOAD_FAST_C_NORC
2	6,346,078	2	INCA_UNPACK_TUPLE_THREE	13	3,151,198	13	FAST_PYFUN_ONE_NORC
3	6,346,078	3	STORE_FAST_B	14	3,151,198	14	INCA_LONG_ADD_NORC_SEC
4	6,346,078	4	STORE_FAST_C	15	3,151,198	15	INCA_LOAD_GLOBAL_NORC
5	6,346,078	5	STORE_FAST_D	16	3,151,198	16	LOAD_FAST_D_NORC
6	6,346,078	6	LOAD_FAST_C_NORC	17	3,151,198	17	FAST_PYFUN_ONE_NORC
7	6,346,078	7	INCA_LOAD_CONST_NORC	18	3,151,198	18	INCA_LONG_SUBTRACT
8	6,346,078	8	COMPARE_OP_NORC	19	3,151,198	19	RETURN_VALUE
9	6,346,078	9	POP_JUMP_IF_FALSE	20	3,194,880	20	LOAD_FAST_B
10	3,151,198	10	LOAD_FAST_B_NORC	21	3,194,880	21	RETURN_VALUE
11	3,151,198	11	INCA_LOAD_GLOBAL_NORC				

Table A.3: Instruction trace and frequency for `check_tree` function of `binarytrees` benchmark.

Interpreter Instruction Scheduling

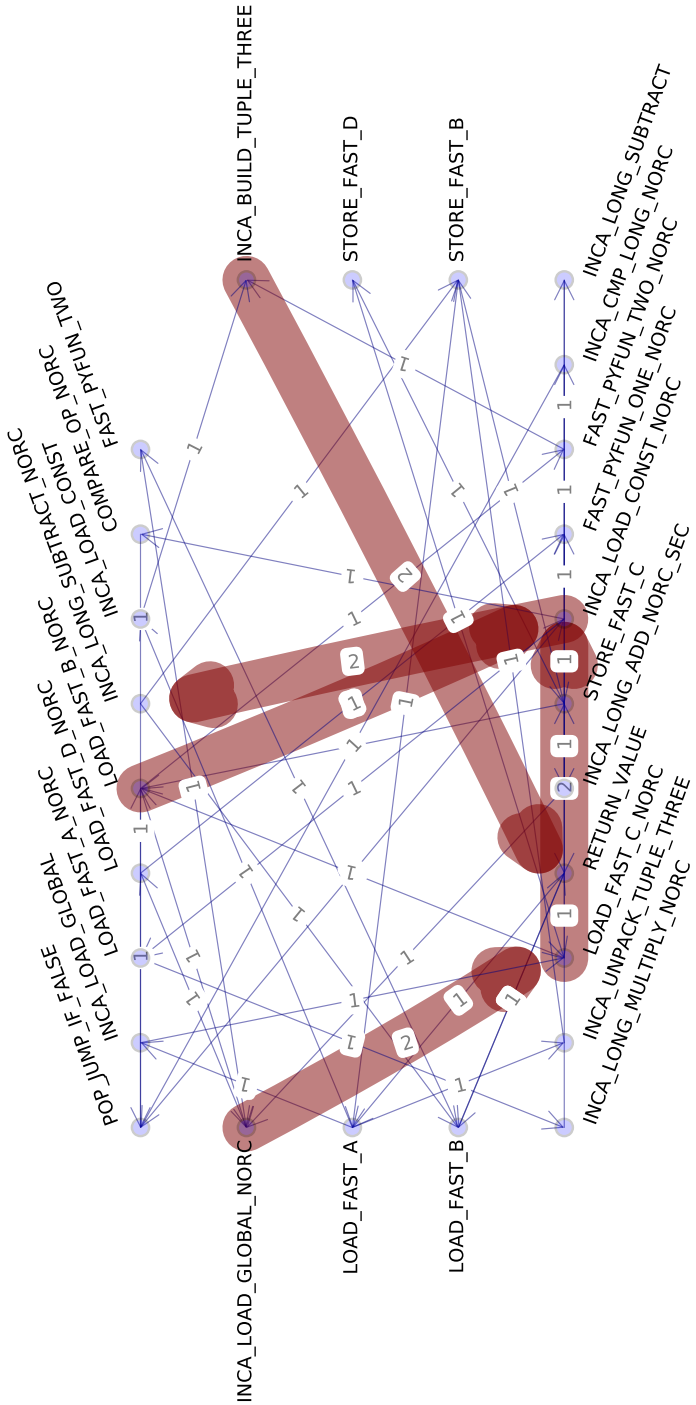


Figure A.1: IIS intermediate graph for `binarytrees` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	INCA_LOAD_CONST_NORC	14	INCA_LONG_ADD_NORC_SEC
2	INCA_LONG_SUBTRACT_NORC	15	INCA_LOAD_GLOBAL_NORC
3	STORE_FAST_B	16	LOAD_FAST_D_NORC
4	STORE_FAST_C	17	INCA_LONG_SUBTRACT
5	LOAD_FAST_B_NORC	18	INCA_LOAD_CONST
6	FAST_PYFUN_TWO_NORC	19	LOAD_FAST_B
7	INCA_BUILD_TUPLE_THREE	20	FAST_PYFUN_TWO
8	RETURN_VALUE	21	STORE_FAST_D
9	LOAD_FAST_A	22	COMPARE_OP_NORC
10	INCA_UNPACK_TUPLE_THREE	23	POP_JUMP_IF_FALSE
11	INCA_LOAD_GLOBAL	24	LOAD_FAST_A_NORC
12	LOAD_FAST_C_NORC	25	INCA_LONG_MULTIPLY_NORC
13	FAST_PYFUN_ONE_NORC	26	INCA_CMP_LONG_NORC

Table A.4: Computed interpreter instruction schedule for the `binarytrees` benchmark (Argument: 14).

A.2 Fannkuch

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	22,806,616	INCA_LOAD_CONST_NORC	9,846,026
2	LOAD_CONST	15,908,857	INCA_LOAD_CONST	6,055,643
3	BINARY_SUBSCR	5,970,294	LOAD_FAST_NORC	5,869,899
4	POP_JUMP_IF_FALSE	5,231,481	INCA_LIST_SUBSCRIPT_NORC	3,951,023
5	STORE_FAST	4,852,554	LOAD_FAST_A_NORC	3,899,599
6	BUILD_SLICE	3,750,098	BUILD_SLICE	3,750,098
7	COMPARE_OP	3,209,377	LOAD_FAST_B_NORC	3,750,041
8	JUMP_ABSOLUTE	2,617,678	INCA_LONG_ADD_NORC	3,723,440
9	STORE_SUBSCR	2,615,691	LOAD_FAST_C	3,462,490
10	INPLACE_ADD	1,991,959	POP_JUMP_IF_FALSE	3,212,977
11	BINARY_ADD	1,734,059	JUMP_ABSOLUTE	2,617,701
12	CALL_FUNCTION	1,251,663	STORE_FAST	2,025,197
13	SETUP_LOOP	1,013,296	STORE_FAST_B	2,018,796
14	INPLACE_SUBTRACT	884,179	POP_JUMP_IF_FALSE_NORC	2,018,526
15	POP_BLOCK	650,423	INCA_LIST_SUBSCRIPT	2,018,525

Table A.5: Comparative dynamic instruction frequency for the `fannkuch` benchmark (Argument: 9).

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	362,880	39	LOAD_FAST_NORC	55	1,731,245	109	LOAD_FAST_C
2	362,880	40	INCA_LOAD_CONST_NORC	56	1,731,245	110	INCA_LOAD_CONST
3	362,880	41	INCA_CMP_LONG_NORC	57	1,731,245	111	LOAD_FAST_B_NORC
4	362,880	42	POP_JUMP_IF_FALSE	58	1,731,245	112	INCA_LOAD_CONST_NORC
5	362,880	59	SETUP_LOOP	59	1,731,245	113	INCA_LONG_ADD_NORC
6	623,530	60	LOAD_FAST_A_NORC	60	1,731,245	114	BUILD_SLICE
7	623,530	61	INCA_LOAD_CONST_NORC	61	1,731,245	115	INCA_LIST_ASS_SUBSCRIPT
8	623,530	62	INCA_CMP_LONG_NORC	62	1,731,245	116	LOAD_FAST_NORC
9	623,530	63	POP_JUMP_IF_FALSE	63	1,731,245	117	INCA_LOAD_CONST_NORC
10	260,650	64	LOAD_FAST_A	64	1,731,245	118	INCA_LONG_ADD_NORC
11	260,650	65	LOAD_FAST_D	65	1,731,245	119	STORE_FAST
12	260,650	66	LOAD_FAST_A_NORC	66	1,731,245	120	LOAD_FAST_C_NORC
13	260,650	67	INCA_LOAD_CONST_NORC	67	1,731,245	121	INCA_LOAD_CONST_NORC
14	260,650	68	INCA_LONG_SUBTRACT_NORC	68	1,731,245	122	INCA_LIST_SUBSCRIPT_NORC
15	260,650	69	INCA_LIST_ASS_SUBSCRIPT	69	1,731,245	123	STORE_FAST_B
16	260,650	70	LOAD_FAST_A_NORC	70	1,731,245	124	JUMP_ABSOLUTE
17	260,650	71	INCA_LOAD_CONST_NORC	71	287,280	125	POP_BLOCK
18	260,650	72	INCA_LONG_SUBTRACT_NORC	72	287,280	126	LOAD_FAST_NORC
19	260,650	73	STORE_FAST_A	73	287,280	127	LOAD_FAST_NORC
20	260,650	74	JUMP_ABSOLUTE	74	287,280	128	INCA_CMP_LONG_NORC
21	362,880	75	POP_BLOCK	75	287,280	129	POP_JUMP_IF_FALSE
22	362,880	76	LOAD_FAST_NORC	76	362,880	134	SETUP_LOOP
23	362,880	77	INCA_LOAD_CONST_NORC	77	623,530	135	LOAD_FAST_A_NORC
24	362,880	78	INCA_LIST_SUBSCRIPT_NORC	78	623,530	136	LOAD_FAST_NORC
25	362,880	79	INCA_LOAD_CONST_NORC	79	623,530	137	INCA_CMP_LONG_NORC
26	362,880	80	INCA_CMP_LONG_NORC_TOS	80	623,530	138	POP_JUMP_IF_FALSE
27	362,880	81	POP_JUMP_IF_FALSE	81	623,529	139	LOAD_FAST_NORC
28	322,560	82	LOAD_FAST_NORC	82	623,529	140	LOAD_FAST_A_NORC
29	322,560	83	LOAD_FAST_NORC	83	623,529	141	LOAD_FAST_NORC
30	322,560	84	INCA_LIST_SUBSCRIPT_NORC	84	623,529	142	INCA_LOAD_CONST_NORC
31	322,560	85	LOAD_FAST_NORC	85	623,529	143	FAST_C_VARARGS_ONE_NORC
32	322,560	86	INCA_CMP_LONG_NORC_TOS	86	623,529	144	FAST_C_VARARGS_TWO_RC_TOS_ONLY
33	322,560	87	POP_JUMP_IF_FALSE	87	623,529	145	POP_TOP
34	287,280	88	LOAD_FAST	88	623,529	146	LOAD_FAST_D_NORC
35	287,280	89	INCA_LOAD_CONST	89	623,529	147	LOAD_FAST_A_NORC
36	287,280	90	INCA_LOAD_CONST	90	623,529	148	DUP_TOPX_NORC
37	287,280	91	BUILD_SLICE	91	623,529	149	INCA_LIST_SUBSCRIPT_NORC
38	287,280	92	INCA_LIST_SUBSCRIPT	92	623,529	150	INCA_LOAD_CONST_NORC
39	287,280	93	STORE_FAST_C	93	623,529	151	INCA_LONG_SUBTRACT_NORC_TOS
40	287,280	94	INCA_LOAD_CONST	94	623,529	152	ROT_THREE
41	287,280	95	STORE_FAST	95	623,529	153	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS
42	287,280	96	LOAD_FAST_C_NORC	96	623,529	154	LOAD_FAST_D_NORC
43	287,280	97	INCA_LOAD_CONST_NORC	97	623,529	155	LOAD_FAST_A_NORC
44	287,280	98	INCA_LIST_SUBSCRIPT_NORC	98	623,529	156	INCA_LIST_SUBSCRIPT_NORC
45	287,280	99	STORE_FAST_B	99	623,529	157	INCA_LOAD_CONST_NORC
46	287,280	100	SETUP_LOOP	100	623,529	158	INCA_CMP_LONG_NORC_TOS
47	2,018,525	101	LOAD_FAST_B_NORC	101	623,529	159	POP_JUMP_IF_FALSE
48	2,018,525	102	POP_JUMP_IF_FALSE_NORC	102	362,879	160	BREAK_LOOP
49	1,731,245	103	LOAD_FAST_C	103	260,650	162	LOAD_FAST_A_NORC
50	1,731,245	104	LOAD_FAST_B	104	260,650	163	INCA_LOAD_CONST_NORC
51	1,731,245	105	INCA_LOAD_CONST	105	260,650	164	INCA_LONG_ADD_NORC
52	1,731,245	106	INCA_LOAD_CONST	106	260,650	165	STORE_FAST_A
53	1,731,245	107	BUILD_SLICE	107	260,650	166	JUMP_ABSOLUTE
54	1,731,245	108	INCA_LIST_SUBSCRIPT	108	362,879	170	JUMP_ABSOLUTE

Table A.6: Instruction trace and frequency for `fannkuch` function of `fannkuch` benchmark.

Interpreter Instruction Scheduling

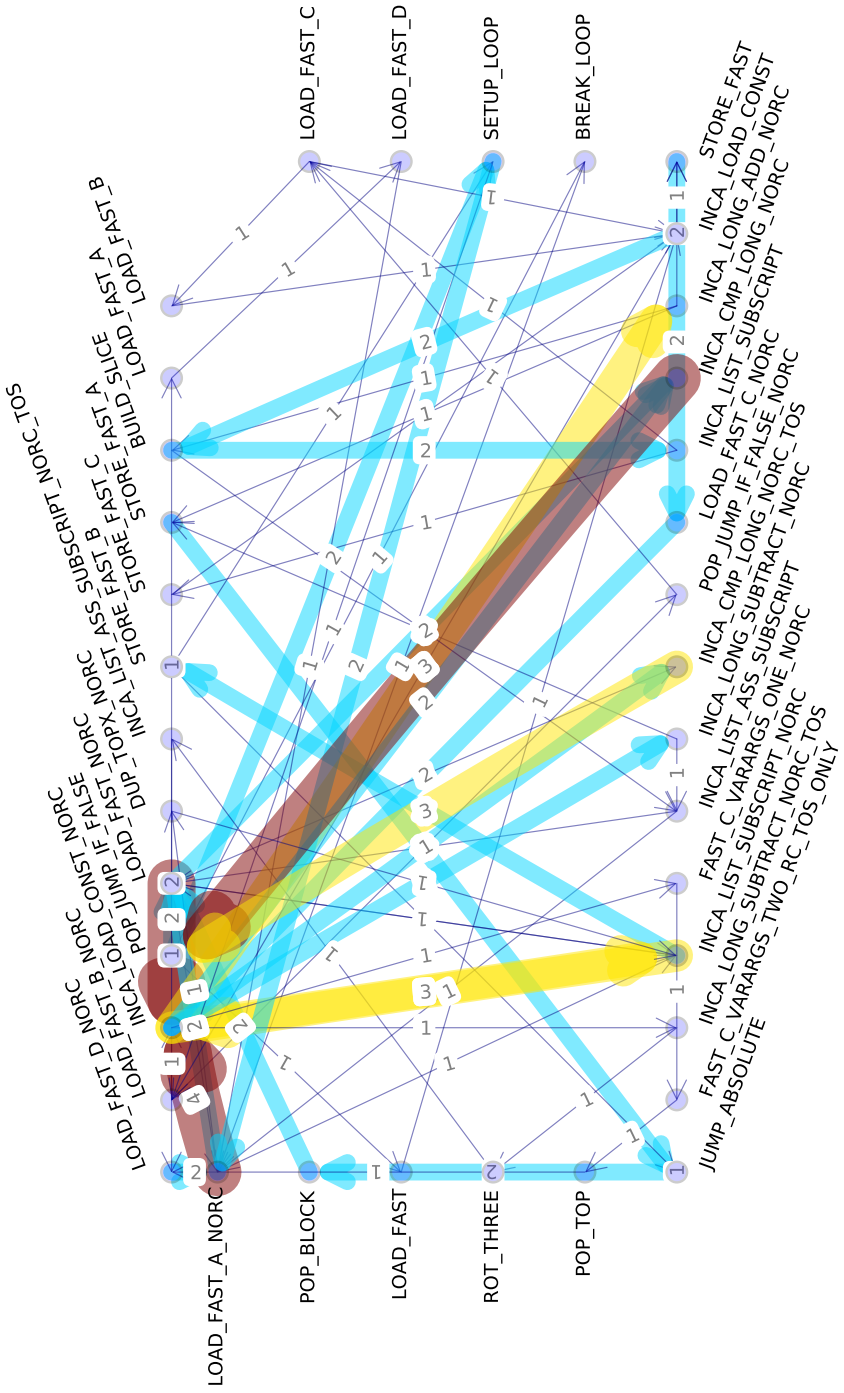


Figure A.2: IIS intermediate graph for `fannkuch` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	INCA_LOAD_CONST_NORC	20	DUP_TOPX_NORC
2	INCA_LONG_ADD_NORC	21	INCA_CMP_LONG_NORC_TOS
3	BUILD_SLICE	22	INCA_LONG_SUBTRACT_NORC
4	INCA_LIST_SUBSCRIPT	23	STORE_FAST_A
5	STORE_FAST_C	24	INCA_LIST_ASS_SUBSCRIPT
6	INCA_LOAD_CONST	25	LOAD_FAST
7	LOAD_FAST_B_NORC	26	BREAK_LOOP
8	POP_JUMP_IF_FALSE_NORC	27	LOAD_FAST_A
9	LOAD_FAST_C	28	LOAD_FAST_D
10	LOAD_FAST_B	29	STORE_FAST
11	INCA_LIST_SUBSCRIPT_NORC	30	LOAD_FAST_C_NORC
12	STORE_FAST_B	31	INCA_LONG_SUBTRACT_NORC_TOS
13	JUMP_ABSOLUTE	32	ROT_THREE
14	POP_BLOCK	33	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS
15	LOAD_FAST_NORC	34	LOAD_FAST_D_NORC
16	INCA_CMP_LONG_NORC	35	FAST_C_VARARGS_ONE_NORC
17	POP_JUMP_IF_FALSE	36	FAST_C_VARARGS_TWO_RC_TOS_ONLY
18	SETUP_LOOP	37	POP_TOP
19	LOAD_FAST_A_NORC		

Table A.7: Computed interpreter instruction schedule for the `fannkuch` benchmark (Argument: 9).

A.3 Fasta

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	2,850,941	LOAD_DEREF_NORC	1,200,000
2	LOAD_DEREF	1,600,000	JUMP_ABSOLUTE	810,342
3	CALL_FUNCTION	825,922	LOAD_FAST_A_NORC	803,358
4	STORE_FAST	816,708	LOAD_FAST_NORC	801,693
5	JUMP_ABSOLUTE	810,319	FOR_ITER_RANGEITER	415,002
6	BINARY_MULTIPLY	801,740	POP_TOP	409,343
7	FOR_ITER	417,739	LOAD_FAST_C_NORC	408,352
8	POP_TOP	409,320	LOAD_FAST_D_NORC	406,669
9	BINARY_ADD	404,212	LOAD_FAST_B_NORC	401,689
10	BINARY_SUBSCR	402,414	STORE_FAST_B	401,687
11	BINARY_MODULO	401,676	INCA_LONG_REMAINDER_NORC_TOS	401,666
12	LIST_APPEND	400,155	INCA_LONG_MULTIPLY_NORC	401,666
13	YIELD_VALUE	400,028	LIST_APPEND	400,155
14	BINARY_TRUE_DIVIDE	400,000	YIELD_VALUE	400,028
15	LOAD_CLOSURE	26,674	STORE_FAST_A	400,025

Table A.8: Comparative dynamic instruction frequency for the `fasta` benchmark. (Argument: 50,000)

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	400,000	8	LOAD_FAST_A_NORC	9	400,000	16	LOAD_FAST_D_NORC
2	400,000	9	LOAD_FAST_NORC	10	400,000	17	LOAD_FAST_A_NORC
3	400,000	10	INCA_LONG_MULTIPLY_NORC	11	400,000	18	INCA_FLOAT_MULTIPLY_NORC
4	400,000	11	LOAD_FAST_NORC	12	400,000	19	LOAD_FAST_C_NORC
5	400,000	12	INCA_LONG_ADD_NORC_TOS	13	400,000	20	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
6	400,000	13	LOAD_FAST_B_NORC	14	400,000	21	YIELD_VALUE
7	400,000	14	INCA_LONG_REMAINDER_NORC_TOS	15	399,999	22	POP_TOP
8	400,000	15	STORE_FAST_A	16	399,999	23	JUMP_ABSOLUTE

Table A.9: Instruction trace and frequency for `genRandom` function of `fasta` benchmark.

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	6,666	1	BUILD_LIST	8	399,960	8	LOAD_DEREF_NORC
2	6,666	2	LOAD_FAST_A	9	399,960	9	FAST_PYFUN_DOCALL_ZERO_NORC
3	406,626	3	FOR_ITER_RANGEITER	10	399,960	10	FAST_C_VARARGS_TWO_RC_TOS_ONLY
4	399,960	4	STORE_FAST_B	11	399,960	11	INCA_LIST_SUBSCRIPT
5	399,960	5	LOAD_DEREF	12	399,960	12	LIST_APPEND
6	399,960	6	LOAD_DEREF_NORC	13	399,960	13	JUMP_ABSOLUTE
7	399,960	7	LOAD_DEREF_NORC	14	6,666	14	RETURN_VALUE

Table A.10: Instruction trace and frequency for the anonymous list comprehension of the `fasta` benchmark.

Interpreter Instruction Scheduling

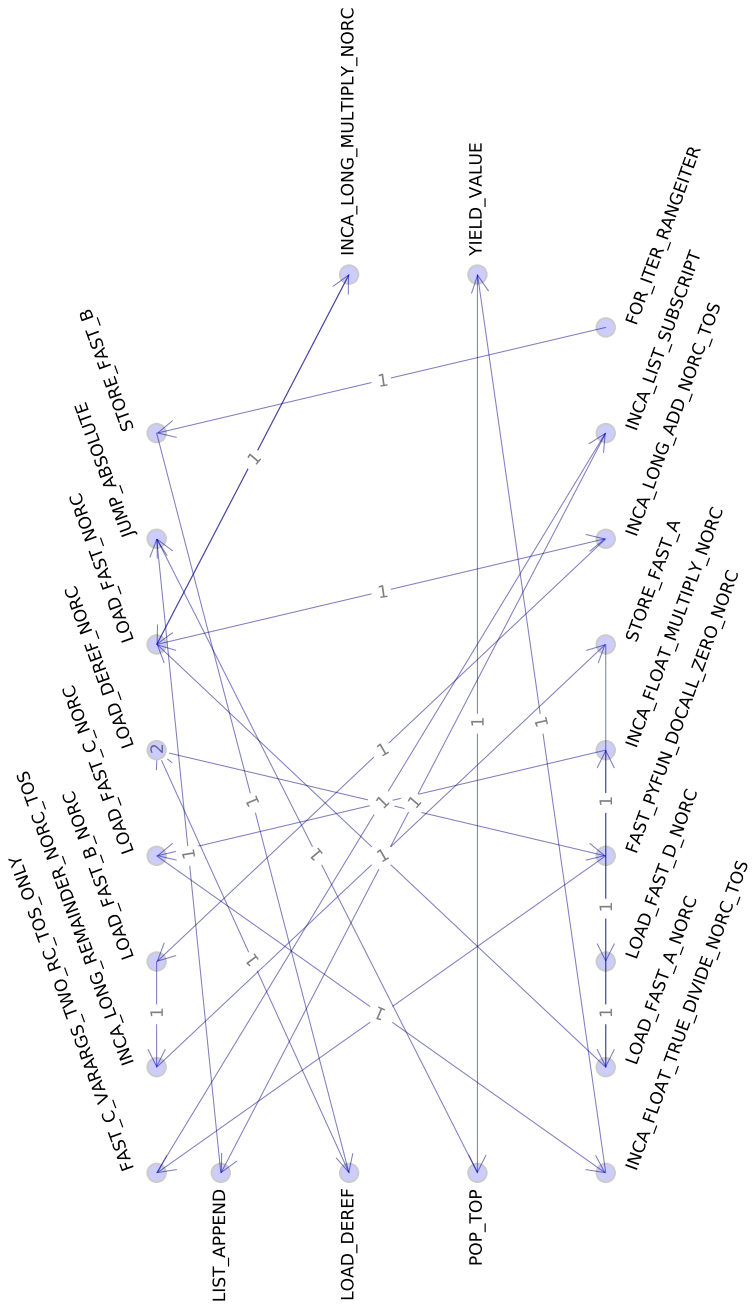


Figure A.3: IIS intermediate graph for `fasta` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	LOAD_DEREF_NORC	12	LOAD_FAST_B_NORC
2	FAST_PYFUN_DOCALL_ZERO_NORC	13	INCA_LONG_REMAINDER_NORC_TOS
3	FAST_C_VARARGS_TWO_RC_TOS_ONLY	14	STORE_FAST_A
4	INCA_LIST_SUBSCRIPT	15	LOAD_FAST_D_NORC
5	LIST_APPEND	16	LOAD_FAST_A_NORC
6	JUMP_ABSOLUTE	17	INCA_FLOAT_MULTIPLY_NORC
7	FOR_ITER_RANGEITER	18	LOAD_FAST_C_NORC
8	STORE_FAST_B	19	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
9	LOAD_DEREF	20	YIELD_VALUE
10	LOAD_FAST_NORC	21	POP_TOP
11	INCA_LONG_ADD_NORC_TOS	22	INCA_LONG_MULTIPLY_NORC

Table A.11: Computed interpreter instruction schedule for the `fasta` benchmark (Argument: 50,000).

A.4 Mandelbrot

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	32,263,553	LOAD_FAST_A_NORC	18,307,122
2	STORE_FAST	13,343,628	INCA_LOAD_CONST_NORC	7,172,150
3	LOAD_CONST	7,492,315	LOAD_FAST_NORC	7,135,378
4	FOR_ITER	6,455,365	FOR_ITER_RANGEITER	6,452,649
5	POP_JUMP_IF_FALSE	6,359,639	STORE_FAST	6,360,001
6	COMPARE_OP	6,356,061	POP_JUMP_IF_FALSE	6,359,662
7	BINARY_ADD	6,354,917	STORE_FAST_A	6,352,374
8	BINARY_MULTIPLY	6,352,944	INCA_COMPLEX_ADD_NORC_TOS	6,352,374
9	CALL_FUNCTION	6,138,113	INCA_COMPLEX_MULTIPLY_NORC	6,102,874
10	JUMP_ABSOLUTE	252,469	FAST_C_ONE_NORC	6,102,374
11	GET_ITER	250,761	LOAD_FAST_D_NORC	6,102,374
12	SETUP_LOOP	250,756	INCA_CMP_FLOAT_NORC_TOS	6,102,374
13	BINARY_SUBTRACT	250,527	LOAD_FAST_C_NORC	568,774
14	BINARY_TRUE_DIVIDE	250,500	INCA_LOAD_CONST	313,007
15	INPLACE_SUBTRACT	219,000	JUMP_ABSOLUTE	252,492

Table A.12: Comparative dynamic instruction frequency for the `mandelbrot` benchmark. (Argument: 500)

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	501	28	FOR_ITER_RANGEITER	44	6,102,374	71	POP_JUMP_IF_FALSE
2	500	29	STORE_FAST	45	150,726	72	BREAK_LOOP
3	500	30	INCA_LOAD_CONST_NORC	46	99,274	75	POP_BLOCK
4	500	31	LOAD_FAST_NORC	47	99,274	76	LOAD_FAST_B_NORC
5	500	32	INCA_COMPLEX_MULTIPLY_NORC	48	99,274	77	INCA_LOAD_CONST_NORC
6	500	33	LOAD_FAST_NORC	49	99,274	78	LOAD_FAST_C_NORC
7	500	34	INCA_COMPLEX_TRUE_DIVIDE_NORC_TOS	50	99,274	79	INCA_LONG_LSHIFT_NORC
8	500	35	INCA_LOAD_CONST_NORC	51	99,274	80	INCA_LONG_ADD_NORC_SEC
9	500	36	INCA_COMPLEX_SUBTRACT_NORC_TOS	52	99,274	81	STORE_FAST_B
10	500	37	STORE_FAST	53	250,000	82	LOAD_FAST_C_NORC
11	500	38	SETUP_LOOP	54	250,000	83	INCA_LOAD_CONST_NORC
12	500	39	LOAD_FAST_NORC	55	250,000	84	INCA_CMP_LONG_NORC
13	500	40	GET_ITER_NORC	56	250,000	85	POP_JUMP_IF_FALSE
14	250,500	41	FOR_ITER_RANGEITER	57	31,000	86	LOAD_FAST_NORC
15	250,000	42	STORE_FAST	58	31,000	87	LOAD_ATTR_NORC
16	250,000	43	INCA_LOAD_CONST	59	31,000	88	LOAD_FAST_B
17	250,000	44	STORE_FAST_A	60	31,000	89	FAST_C_ONE
18	250,000	45	INCA_LOAD_CONST_NORC	61	31,000	90	POP_TOP
19	250,000	46	LOAD_FAST_NORC	62	31,000	91	INCA_LOAD_CONST
20	250,000	47	INCA_FLOAT_MULTIPLY_NORC	63	31,000	92	STORE_FAST_C
21	250,000	48	LOAD_FAST_NORC	64	31,000	93	INCA_LOAD_CONST
22	250,000	49	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS	65	31,000	94	STORE_FAST_B
23	250,000	50	INCA_LOAD_CONST_NORC	66	31,000	95	JUMP_ABSOLUTE
24	250,000	51	INCA_FLOAT_SUBTRACT_NORC_TOS	67	219,000	96	LOAD_FAST_C_NORC
25	250,000	52	LOAD_FAST_NORC	68	219,000	97	INCA_LOAD_CONST_NORC
26	250,000	53	INCA_COMPLEX_ADD_NORC_TOS	69	219,000	98	INCA_LONG_SUBTRACT_NORC
27	250,000	54	STORE_FAST_D	70	219,000	99	STORE_FAST_C
28	250,000	55	SETUP_LOOP	71	219,000	100	JUMP_ABSOLUTE
29	250,000	56	LOAD_FAST_NORC	72	500	101	POP_BLOCK
30	250,000	57	GET_ITER_NORC	73	500	102	LOAD_FAST_C_NORC
31	6,201,648	58	FOR_ITER_RANGEITER	74	500	103	INCA_LOAD_CONST_NORC
32	6,102,374	59	STORE_FAST	75	500	104	INCA_CMP_LONG_NORC
33	6,102,374	60	LOAD_FAST_A_NORC	76	500	105	POP_JUMP_IF_FALSE
34	6,102,374	61	LOAD_FAST_A_NORC	77	500	106	LOAD_FAST_NORC
35	6,102,374	62	INCA_COMPLEX_MULTIPLY_NORC	78	500	107	LOAD_ATTR_NORC
36	6,102,374	63	LOAD_FAST_D_NORC	79	500	108	LOAD_FAST_B
37	6,102,374	64	INCA_COMPLEX_ADD_NORC_TOS	80	500	109	FAST_C_ONE
38	6,102,374	65	STORE_FAST_A	81	500	110	POP_TOP
39	6,102,374	66	LOAD_FAST_NORC	82	500	111	INCA_LOAD_CONST
40	6,102,374	67	LOAD_FAST_A_NORC	83	500	112	STORE_FAST_C
41	6,102,374	68	FAST_C_ONE_NORC	84	500	113	INCA_LOAD_CONST
42	6,102,374	69	INCA_LOAD_CONST_NORC	85	500	114	STORE_FAST_B
43	6,102,374	70	INCA_CMP_FLOAT_NORC_TOS	86	500	115	JUMP_ABSOLUTE

Table A.13: Instruction trace and frequency for the `mandelbrot` function of the `mandelbrot` benchmark.

Interpreter Instruction Scheduling

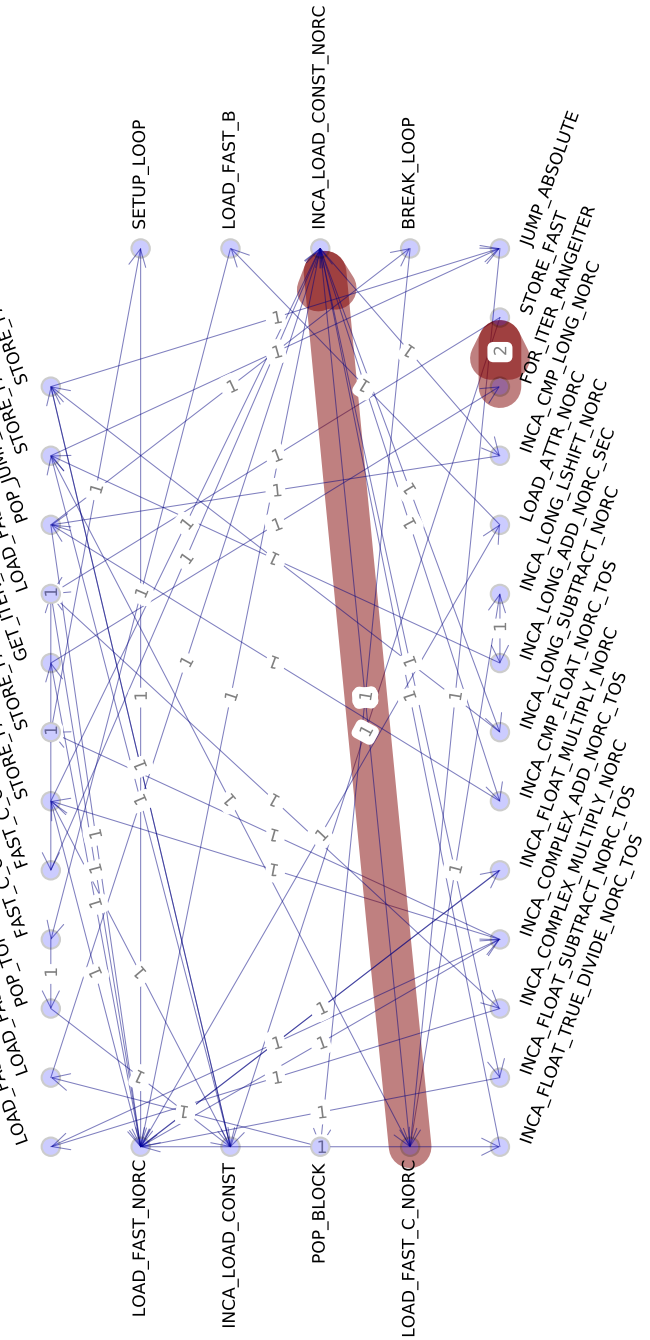


Figure A.4: IIS intermediate graph for `mandelbrot` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	LOAD_FAST_NORC	18	BREAK_LOOP
2	LOAD_FAST_A_NORC	19	POP_BLOCK
3	INCA_COMPLEX_MULTIPLY_NORC	20	LOAD_FAST_B_NORC
4	LOAD_FAST_D_NORC	21	LOAD_ATTR_NORC
5	INCA_COMPLEX_ADD_NORC_TOS	22	LOAD_FAST_B
6	STORE_FAST_A	23	FAST_C_ONE
7	INCA_LOAD_CONST_NORC	24	POP_TOP
8	INCA_LONG_SUBTRACT_NORC	25	INCA_FLOAT_MULTIPLY_NORC
9	STORE_FAST_C	26	STORE_FAST_D
10	JUMP_ABSOLUTE	27	SETUP_LOOP
11	LOAD_FAST_C_NORC	28	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
12	INCA_LONG_LSHIFT_NORC	29	GET_ITER_NORC
13	INCA_LONG_ADD_NORC_SEC	30	FOR_ITER_RANGEITER
14	STORE_FAST_B	31	STORE_FAST
15	INCA_LOAD_CONST	32	INCA_CMP_FLOAT_NORC_TOS
16	INCA_CMP_LONG_NORC	33	INCA_FLOAT_SUBTRACT_NORC_TOS
17	POP_JUMP_IF_FALSE	34	FAST_C_ONE_NORC

Table A.14: Computed interpreter instruction schedule for the `mandelbrot` benchmark (Argument: 500).

A.5 Nbody

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	19,819,638	INCA_LOAD_CONST_NORC	9,000,317
2	LOAD_CONST	9,007,540	INCA_LIST_SUBSCRIPT_NORC	8,500,260
3	BINARY_SUBSCR	8,501,013	LOAD_FAST_NORC	7,300,084
4	BINARY_MULTIPLY	6,750,233	LOAD_FAST_A_NORC	5,000,111
5	STORE_FAST	4,306,788	INCA_FLOAT_MULTIPLY_NORC	4,500,060
6	STORE_SUBSCR	3,750,270	ROT_THREE	3,750,027
7	ROT_THREE	3,750,027	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	3,750,000
8	DUP_TOPX	3,750,000	DUP_TOPX_NORC	3,750,000
9	INPLACE_ADD	2,250,044	LOAD_FAST_B_NORC	3,500,160
10	BINARY_SUBTRACT	1,500,087	INCA_FLOAT_ADD	3,250,060
11	INPLACE_SUBTRACT	1,500,035	INCA_FLOAT_SUBTRACT	3,000,095
12	BINARY_ADD	1,002,608	STORE_FAST	2,056,696
13	FOR_ITER	902,786	LOAD_FAST_D_NORC	2,000,041
14	JUMP_ABSOLUTE	802,019	LOAD_FAST_C_NORC	2,000,041
15	UNPACK_SEQUENCE	500,266	INCA_FLOAT_MULTIPLY_NORC_SEC	1,250,010

Table A.15: Comparative dynamic instruction frequency for the `nbody` benchmark. (Argument: 50,000)

Interpreter Instruction Scheduling

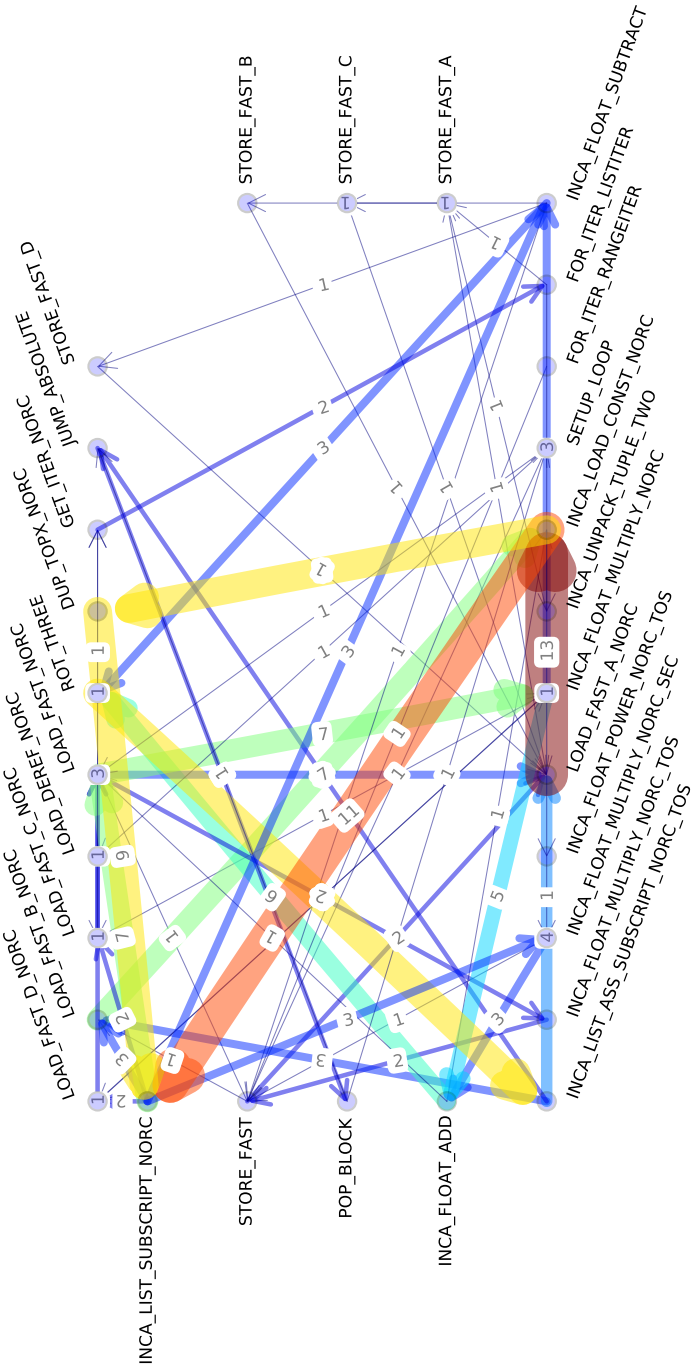


Figure A.5: IIS intermediate graph for `nbody` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	INCA_LOAD_CONST_NORC	16	LOAD_DEREF_NORC
2	INCA_LIST_SUBSCRIPT_NORC	17	GET_ITER_NORC
3	LOAD_FAST_NORC	18	FOR_ITER_LISTITER
4	INCA_FLOAT_MULTIPLY_NORC	19	STORE_FAST_A
5	INCA_FLOAT_ADD	20	STORE_FAST_B
6	ROT_THREE	21	STORE_FAST_D
7	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	22	INCA_FLOAT_MULTIPLY_NORC_SEC
8	LOAD_FAST_A_NORC	23	JUMP_ABSOLUTE
9	DUP_TOPX_NORC	24	POP_BLOCK
10	LOAD_FAST_B_NORC	25	LOAD_FAST_C_NORC
11	INCA_FLOAT_SUBTRACT	26	LOAD_FAST_D_NORC
12	STORE_FAST_C	27	INCA_UNPACK_TUPLE_TWO
13	INCA_FLOAT_MULTIPLY_NORC_TOS	28	INCA_FLOAT_POWER_NORC_TOS
14	STORE_FAST	29	FOR_ITER_RANGEITER
15	SETUP_LOOP		

Table A.16: Computed interpreter instruction schedule for the `nbody` benchmark (Argument: 50,000).

A.6 Spectralnorm

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	64,085,676	LOAD_FAST_A_NORC	25,601,220
2	BINARY_ADD	32,002,543	INCA_LOAD_CONST_NORC	19,200,002
3	LOAD_CONST	25,623,176	LOAD_FAST_NORC	12,816,822
4	STORE_FAST	19,240,312	LOAD_FAST_B_NORC	12,816,051
5	BINARY_MULTIPLY	12,800,871	INCA_FLOAT_ADD_NORC_TOS	12,800,000
6	BINARY_TRUE_DIVIDE	12,800,001	INCA_LONG_ADD_NORC	12,800,000
7	CALL_FUNCTION	6,468,380	JUMP_ABSOLUTE	6,418,402
8	FOR_ITER	6,435,168	FOR_ITER_ZIP	6,416,401
9	JUMP_ABSOLUTE	6,418,379	STORE_FAST_A	6,416,400
10	RETURN_VALUE	6,401,632	INCA_LOAD_CONST	6,416,015
11	INPLACE_ADD	6,400,834	RETURN_VALUE	6,401,632
12	UNPACK_SEQUENCE	6,400,661	INCA_FLOAT_ADD_NORC_SEC	6,400,800
13	LOAD_GLOBAL	52,564	STORE_FAST_D	6,400,401
14	LOAD_ATTR	20,453	LOAD_FAST_D_NORC	6,400,401
15	POP_TOP	16,985	INCA_UNPACK_TUPLE_TWO	6,400,400

Table A.17: Comparative dynamic instruction frequency for the `spectralnorm` benchmark. (Argument: 400)

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	6,400,000	1	INCA_LOAD_CONST	10	6,400,000	10	INCA_LONG_MULTIPLY
2	6,400,000	2	LOAD_FAST_A_NORC	11	6,400,000	11	INCA_LOAD_CONST_NORC
3	6,400,000	3	LOAD_FAST_B_NORC	12	6,400,000	12	INCA_LONG_TRUE_DIVIDE_NORC_TOS
4	6,400,000	4	INCA_LONG_ADD_NORC	13	6,400,000	13	LOAD_FAST_A_NORC
5	6,400,000	5	LOAD_FAST_A_NORC	14	6,400,000	14	INCA_FLOAT_ADD_NORC_TOS
6	6,400,000	6	LOAD_FAST_B_NORC	15	6,400,000	15	INCA_LOAD_CONST_NORC
7	6,400,000	7	INCA_LONG_ADD_NORC	16	6,400,000	16	INCA_FLOAT_ADD_NORC_TOS
8	6,400,000	8	INCA_LOAD_CONST_NORC	17	6,400,000	17	INCA_FLOAT_TRUE_DIVIDE
9	6,400,000	9	INCA_LONG_ADD_NORC_TOS	18	6,400,000	18	RETURN_VALUE

Table A.18: Instruction trace and frequency for the `eval_A` function of the `spectralnorm` benchmark.

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	8,020	12	FOR_ITER_RANGEITER	19	3,200,000	30	LOAD_FAST_A_NORC
2	8,000	13	STORE_FAST	20	3,200,000	31	LOAD_FAST_NORC
3	8,000	14	INCA_LOAD_CONST	21	3,200,000	32	LOAD_FAST_NORC
4	8,000	15	STORE_FAST_A	22	3,200,000	33	LOAD_FAST_C_NORC
5	8,000	16	SETUP_LOOP	23	3,200,000	34	FAST_PYFUN_TWO_NORC
6	8,000	17	INCA_LOAD_GLOBAL	24	3,200,000	35	LOAD_FAST_D_NORC
7	8,000	18	INCA_LOAD_GLOBAL_NORC	25	3,200,000	36	INCA_FLOAT_MULTIPLY_NORC_TOS
8	8,000	19	INCA_LOAD_GLOBAL_NORC	26	3,200,000	37	INCA_FLOAT_ADD_NORC_SEC
9	8,000	20	LOAD_FAST_B_NORC	27	3,200,000	38	STORE_FAST_A
10	8,000	21	FAST_C_ONE_NORC	28	3,200,000	39	JUMP_ABSOLUTE
11	8,000	22	FAST_PYFUN_DOCALL_ONE_RC_TOS_ONLY	29	8,000	40	POP_BLOCK
12	8,000	23	LOAD_FAST_B	30	8,000	41	LOAD_FAST_NORC
13	8,000	24	FAST_PYFUN_DOCALL_TWO	31	8,000	42	LOAD_ATTR_NORC
14	8,000	25	GET_ITER	32	8,000	43	LOAD_FAST_A
15	3,208,000	26	FOR_ITER_ZIP	33	8,000	44	FAST_C_ONE
16	3,200,000	27	INCA_UNPACK_TUPLE_TWO	34	8,000	45	POP_TOP
17	3,200,000	28	STORE_FAST_C	35	8,000	46	JUMP_ABSOLUTE
18	3,200,000	29	STORE_FAST_D				

Table A.19: Instruction trace and frequency for the `eval_A_times_u` function of the `spectralnorm` benchmark.

Interpreter Instruction Scheduling

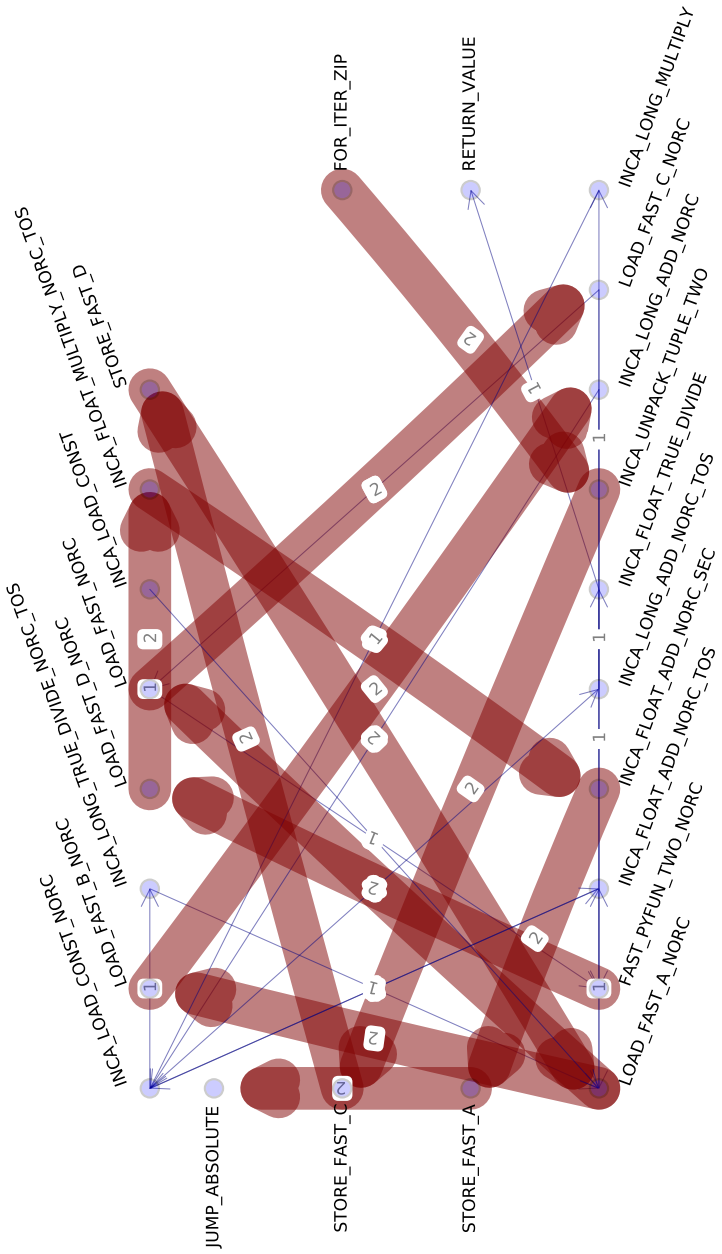


Figure A.6: IIS intermediate graph for `spectralnorm` benchmark.

Schedule

Rank	Instruction	Rank	Instruction
1	LOAD_FAST_A_NORC	13	LOAD_FAST_C_NORC
2	LOAD_FAST_B_NORC	14	FAST_PYFUN_TWO_NORC
3	INCA_LONG_ADD_NORC	15	LOAD_FAST_D_NORC
4	INCA_LOAD_CONST_NORC	16	INCA_FLOAT_MULTIPLY_NORC_TOS
5	INCA_FLOAT_ADD_NORC_TOS	17	INCA_FLOAT_ADD_NORC_SEC
6	INCA_FLOAT_TRUE_DIVIDE	18	STORE_FAST_A
7	RETURN_VALUE	19	JUMP_ABSOLUTE
8	FOR_ITER_ZIP	20	INCA_LONG_TRUE_DIVIDE_NORC_TOS
9	INCA_UNPACK_TUPLE_TWO	21	INCA_LONG_ADD_NORC_TOS
10	STORE_FAST_C	22	INCA_LONG_MULTIPLY
11	STORE_FAST_D	23	INCA_LOAD_CONST
12	LOAD_FAST_NORC		

Table A.20: Computed interpreter instruction schedule for the `spectralnorm` benchmark (Argument: 400).

A.7 Django

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	9,184,485	INCA_LOAD_GLOBAL_NORC	4,397,549
2	LOAD_GLOBAL	5,625,935	LOAD_FAST	3,533,608
3	CALL_FUNCTION	4,594,435	LOAD_FAST_A_NORC	2,990,177
4	LOAD_CONST	2,549,065	POP_JUMP_IF_FALSE	1,788,535
5	POP_JUMP_IF_FALSE	2,332,208	FAST_C_VARARGS_TWO_NORC	1,623,574
6	LOAD_ATTR	2,230,628	RETURN_VALUE	1,445,828
7	STORE_FAST	2,055,285	LOAD_ATTR	1,255,999
8	RETURN_VALUE	1,445,800	STORE_FAST	1,245,146
9	FOR_ITER	1,421,961	CALL_FUNCTION	1,147,164
10	POP_BLOCK	1,088,246	POP_BLOCK	1,088,249
11	POP_JUMP_IF_TRUE	1,085,255	POP_JUMP_IF_TRUE	1,085,382
12	JUMP_ABSOLUTE	933,943	LOAD_ATTR_NORC	974,795
13	JUMP_FORWARD	752,603	JUMP_ABSOLUTE	934,068
14	SETUP_EXCEPT	746,021	INCA_LOAD_CONST	877,141
15	STORE_SUBSCR	682,121	LOAD_CONST	862,260

Table A.21: Comparative dynamic instruction frequency for the `django` benchmark.

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	473,360	1	LOAD_DEREF_NORC	21	67,375	44	LOAD_FAST_A_NORC
2	473,360	2	POP_JUMP_IF_FALSE_NORC	22	67,375	45	INCA_LOAD_GLOBAL_NORC
3	473,360	21	SETUP_EXCEPT	23	67,375	46	FAST_C_VARARGS_TWO_NORC
4	473,360	22	INCA_LOAD_GLOBAL_NORC	24	67,375	47	POP_JUMP_IF_FALSE
5	473,360	23	LOAD_FAST_A_NORC	25	67,375	55	INCA_LOAD_GLOBAL_NORC
6	473,360	24	INCA_LOAD_GLOBAL_NORC	26	67,375	56	LOAD_FAST_A_NORC
7	473,360	25	FAST_C_VARARGS_TWO_NORC	27	67,375	57	FAST_PYFUN_DOCALL_ONE_NORC
8	473,360	26	POP_JUMP_IF_TRUE	28	67,375	58	STORE_FAST_A
9	67,375	27	INCA_LOAD_GLOBAL_NORC	29	67,375	59	JUMP_FORWARD
10	67,375	28	LOAD_FAST_A_NORC	30	67,375	68	POP_BLOCK
11	67,375	29	INCA_LOAD_CONST_NORC	31	67,375	69	JUMP_ABSOLUTE
12	67,375	30	FAST_C_VARARGS_TWO_NORC	32	405,985	101	INCA_LOAD_GLOBAL_NORC
13	67,375	31	POP_JUMP_IF_FALSE	33	405,985	102	LOAD_FAST_A_NORC
14	67,375	37	SETUP_EXCEPT	34	405,985	103	INCA_LOAD_GLOBAL_NORC
15	67,375	38	INCA_LOAD_GLOBAL_NORC	35	405,985	104	FAST_C_VARARGS_TWO_NORC
16	67,375	39	LOAD_ATTR_NORC	36	405,985	105	POP_JUMP_IF_TRUE
17	67,375	40	LOAD_CONST_NORC	37	473,360	113	POP_BLOCK
18	67,375	41	INCA_CMP_LONG_NORC_TOS	38	473,360	114	JUMP_FORWARD
19	67,375	42	POP_JUMP_IF_FALSE	39	473,360	138	LOAD_FAST_A
20	67,375	43	INCA_LOAD_GLOBAL_NORC	40	473,360	139	RETURN_VALUE

Table A.22: Instruction trace and frequency for the `force_unicode` function of the `django` benchmark.

Interpreter Instruction Scheduling

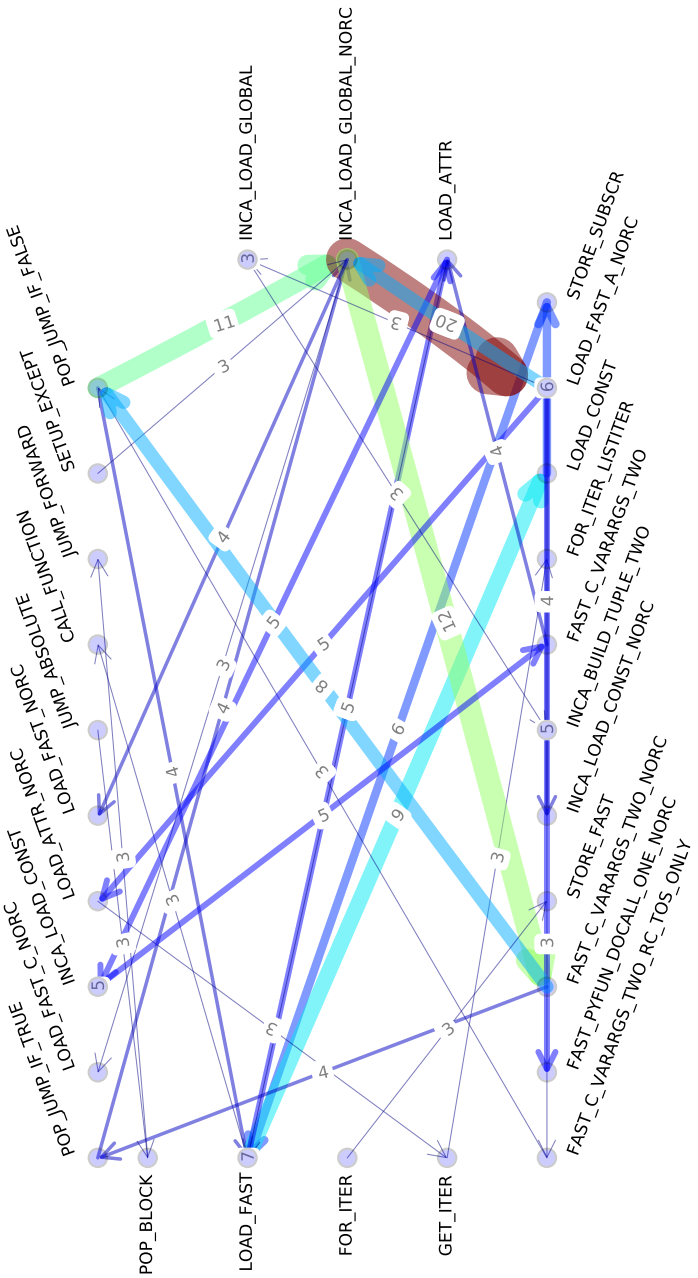


Figure A.7: IIS intermediate graph for `django` benchmark (*without* edges of weight ≤ 2).

Schedule

Rank	Instruction	Rank	Instruction
1	INCA_LOAD_GLOBAL_NORC	37	LOAD_CONST_NORC
2	LOAD_FAST_A_NORC	38	INCA_CMP_LONG_NORC_TOS
3	FAST_PYFUN_DOCALL_ONE_NORC	39	INCA_LIST_SUBSCRIPT_NORC
4	STORE_FAST_A	40	CALL_FUNCTION_RC_TOS_ONLY
5	POP_BLOCK	41	GET_ITER_NORC
6	JUMP_FORWARD	42	BINARY_SUBTRACT
7	LOAD_FAST_B_NORC	43	COMPARE_OP
8	LOAD_ATTR_NORC	44	LIST_APPEND
9	GET_ITER	45	INCA_LOAD_GLOBAL
10	FOR_ITER_LISTITER	46	INCA_BUILD_TUPLE_TWO
11	STORE_FAST_C	47	FAST_C_VARARGS_TWO_RC_TOS_ONLY
12	SETUP_EXCEPT	48	LOAD_FAST_C_NORC
13	LOAD_FAST_NORC	49	CALL_FUNCTION_NORC
14	INCA_LOAD_CONST_NORC	50	LOAD_FAST_C
15	FAST_C_VARARGS_THREE_NORC	51	COMPARE_OP_NORC_TOS
16	POP_JUMP_IF_FALSE	52	POP_JUMP_IF_FALSE_NORC
17	LOAD_FAST	53	LOAD_DEREF
18	LOAD_CONST	54	LOAD_FAST_A
19	STORE_SUBSCR	55	LOAD_FAST_B
20	SETUP_LOOP	56	CALL_FUNCTION_VAR_KW
21	LOAD_FAST_D_NORC	57	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS
22	FAST_C_VARARGS_TWO_NORC	58	CALL_FUNCTION_VAR
23	POP_JUMP_IF_TRUE	59	BINARY_SUBSCR_NORC
24	LOAD_ATTR	60	COMPARE_OP_NORC
25	INCA_LOAD_CONST	61	INCA_DICT_SUBSCRIPT_NORC
26	FAST_C_VARARGS_TWO	62	UNARY_NOT
27	FAST_PYFUN_ONE_RC_TOS_ONLY	63	BUILD_LIST
28	RETURN_VALUE	64	STORE_FAST_B
29	FOR_ITER	65	FAST_C_ZERO_RC_TOS_ONLY
30	STORE_FAST	66	FAST_PYFUN_DOCALL_ONE_RC_TOS_ONLY
31	LOAD_GLOBAL	67	BINARY_ADD
32	CALL_FUNCTION	68	UNPACK_SEQUENCE
33	POP_TOP	69	FOR_ITER_TUPLEITER
34	JUMP_ABSOLUTE	70	BINARY_SUBSCR
35	LOAD_FAST_D	71	INCA_UNPACK_TUPLE_TWO
36	FAST_PYMETH_ONE	72	LOAD_DEREF_NORC

Table A.23: Computed interpreter instruction schedule for the `django` benchmark.

A.8 AI

Dynamic Bytecode Frequencies

No.	Standard Instruction	Frequency	Optimized Instruction	Frequency
1	LOAD_FAST	6,219,727	LOAD_FAST_B_NORC	2,316,704
2	BINARY_SUBSCR	2,481,525	JUMP_ABSOLUTE	1,607,889
3	FOR_ITER	1,868,739	FOR_ITER_LISTITER	1,539,966
4	STORE_FAST	1,727,699	POP_TOP	1,533,860
5	JUMP_ABSOLUTE	1,607,866	YIELD_VALUE	1,455,306
6	POP_TOP	1,533,837	STORE_FAST_B	1,371,449
7	LOAD_CONST	1,523,192	LOAD_DEREF_NORC	1,364,372
8	YIELD_VALUE	1,455,306	INCA_TUPLE_SUBSCRIPT_NORC	1,364,368
9	LOAD_DEREF	1,364,559	LOAD_FAST_A_NORC	1,294,886
10	BINARY_ADD	1,072,467	INCA_LOAD_CONST_NORC	742,294
11	CALL_FUNCTION	744,620	LOAD_FAST_B	692,775
12	STORE_SUBSCR	693,327	LOAD_FAST_C_NORC	692,769
13	LOAD_GLOBAL	556,424	INCA_LOAD_CONST	651,823
14	BUILD_SLICE	522,004	INCA_LONG_ADD_NORC_TOS	645,135
15	POP_JUMP_IF_FALSE	437,200	INCA_LIST_SUBSCRIPT_NORC	643,272

Table A.24: Comparative dynamic instruction frequency for the `ai` benchmark.

No.	Freq.	Off.	Instruction	No.	Freq.	Off.	Instruction
1	80,642	12	FOR_ITER	20	4,226	31	INCA_LOAD_GLOBAL_NORC
2	80,640	13	STORE_DEREF	21	4,226	32	LOAD_CLOSURE
3	80,640	14	LOAD_FAST_A	22	4,226	33	BUILD_TUPLE
4	80,640	15	INCA_LOAD_GLOBAL_NORC	23	4,226	34	INCA_LOAD_CONST
5	80,640	16	INCA_LOAD_GLOBAL_NORC	24	4,226	35	MAKE_CLOSURE
6	80,640	17	LOAD_CLOSURE	25	4,226	36	LOAD_FAST_B_NORC
7	80,640	18	BUILD_TUPLE	26	4,226	37	GET_ITER_NORC
8	80,640	19	INCA_LOAD_CONST	27	4,226	38	FAST_CALL_GENERATOR_ONE
9	80,640	20	MAKE_CLOSURE	28	4,226	39	FAST_PYFUN_DOCALL_ONE_RC_TOS_ONLY
10	80,640	21	LOAD_FAST_B_NORC	29	4,226	40	FAST_C_ONE_RC_TOS_ONLY
11	80,640	22	GET_ITER_NORC	30	4,226	41	INCA_CMP_LONG
12	80,640	23	FAST_CALL_GENERATOR_ONE	31	4,226	42	JUMP_FORWARD
13	80,640	24	FAST_PYFUN_DOCALL_ONE_RC_TOS_ONLY	32	76,414	43	ROT_TWO
14	80,640	25	FAST_C_ONE_RC_TOS_ONLY	33	76,414	44	POP_TOP
15	80,640	26	DUP_TOP	34	80,640	45	POP_JUMP_IF_FALSE
16	80,640	27	ROT_THREE	35	184	46	LOAD_DEREF
17	80,640	28	INCA_CMP_LONG	36	184	47	YIELD_VALUE
18	80,640	29	JUMP_IF_FALSE_OR_POP	37	184	48	POP_TOP
19	4,226	30	INCA_LOAD_GLOBAL_NORC	38	184	49	JUMP_ABSOLUTE

Table A.25: Instruction trace and frequency for the `n_queens` function of the `ai` benchmark.

Interpreter Instruction Scheduling

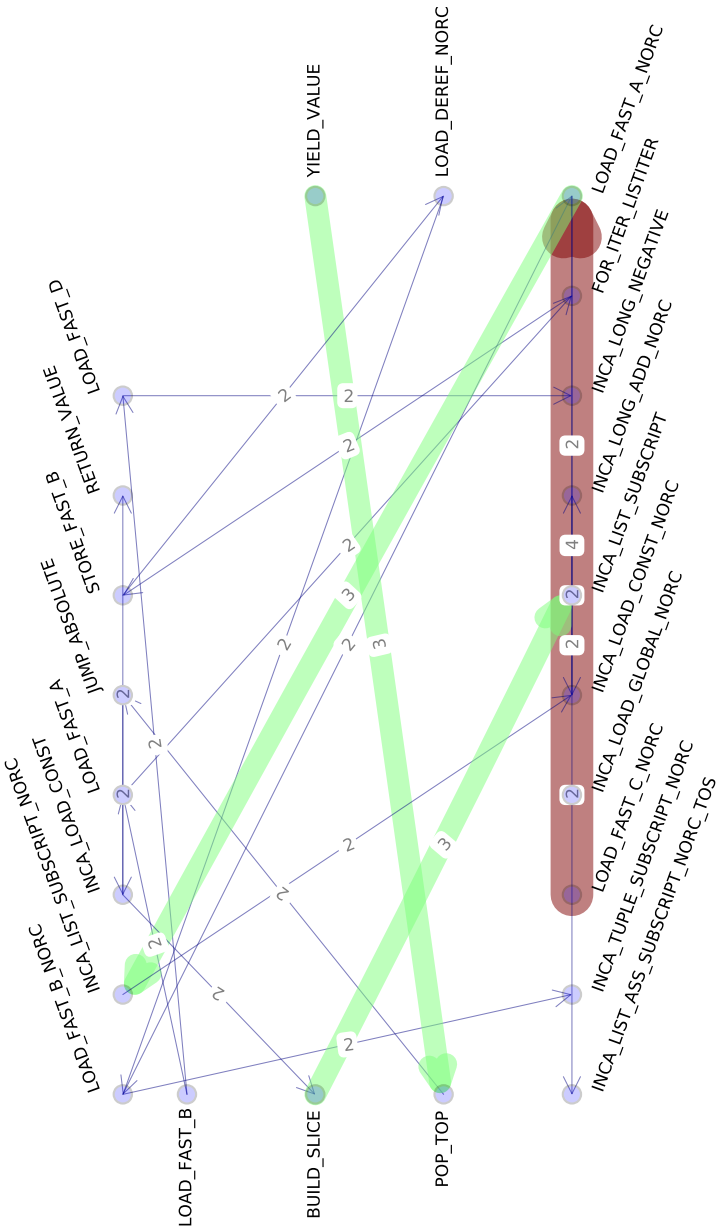


Figure A.8: IIS intermediate graph for `ai` benchmark (*without* edges of weight ≤ 1).

Schedule

Rank	Instruction	Rank	Instruction
1	LOAD_FAST_A_NORC	26	INCA_LOAD_GLOBAL_NORC
2	INCA_LIST_SUBSCRIPT_NORC	27	LOAD_CLOSURE
3	INCA_LOAD_CONST_NORC	28	BUILD_TUPLE
4	INCA_LONG_ADD_NORC	29	LOAD_CONST
5	BUILD_SLICE	30	MAKE_CLOSURE
6	INCA_LIST_SUBSCRIPT	31	INCA_CMP_LONG_NORC_TOS
7	LOAD_FAST_B_NORC	32	POP_JUMP_IF_FALSE
8	INCA_TUPLE_SUBSCRIPT_NORC	33	LOAD_FAST_NORC
9	YIELD_VALUE	34	POP_JUMP_IF_FALSE_NORC
10	POP_TOP	35	SETUP_LOOP
11	JUMP_ABSOLUTE	36	DUP_TOPX_NORC
12	INCA_LOAD_CONST	37	INCA_LONG_SUBTRACT_NORC
13	RETURN_VALUE	38	FAST_PYFUN_DOCALL_ONE_NORC
14	LOAD_FAST_A	39	FAST_PYFUN_DOCALL_ONE_RC_TOS_ONLY
15	FOR_ITER_LISTITER	40	GET_ITER
16	STORE_FAST_B	41	FAST_CALL_GENERATOR_ONE
17	LOAD_DEREF_NORC	42	FOR_ITER_LISTREVITER
18	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	43	STORE_FAST_A
19	LOAD_FAST_C_NORC	44	BINARY_ADD
20	INCA_LONG_SUBTRACT_NORC_TOS	45	LOAD_FAST
21	ROT_THREE	46	BREAK_LOOP
22	LOAD_FAST_B	47	INCA_LONG_ADD_NORC_TOS
23	LOAD_FAST_D	48	ROT_TWO
24	INCA_LONG_NEGATIVE	49	STORE_FAST_D
25	INCA_LIST_ASS_SUBSCRIPT		

Table A.26: Computed interpreter instruction schedule for the `ai` benchmark.

A.9 Reference Count Quickening Details

Benchmark	Reduction	
	Increment	Decrement
ai-1	0.7789	0.7885
binarytrees-14	0.6906	0.7664
django-1	0.8416	0.8426
fannkuch-9	0.7212	0.7279
fasta-50000	0.6351	0.6544
mandelbrot-500	0.2576	0.4596
nbody-50000	0.4218	0.5250
spectralnorm-400	0.5283	0.6976

Table A.27: Relative reduction of reference count operations per benchmark.

Benchmark	RC Operations per Bytecode		
	Standard	Optimized	Ratio
ai-1	2.7479	2.1543	0.7840
binarytrees-14	1.9073	1.3929	0.7303
django-1	3.2832	2.7649	0.8422
fannkuch-9	2.6886	1.9483	0.7247
fasta-50000	2.2658	1.4635	0.6459
mandelbrot-500	1.3523	0.5064	0.3745
nbody-50000	2.0210	0.9670	0.4785
spectralnorm-400	1.7604	1.1049	0.6276
Average	2.2533	1.5378	0.6825

Table A.28: Number of reference count operations per bytecode per benchmark.

Appendix B

Comparison of Benchmark Results

Benchmark Run-time Results for Intel Nehalem i7-920

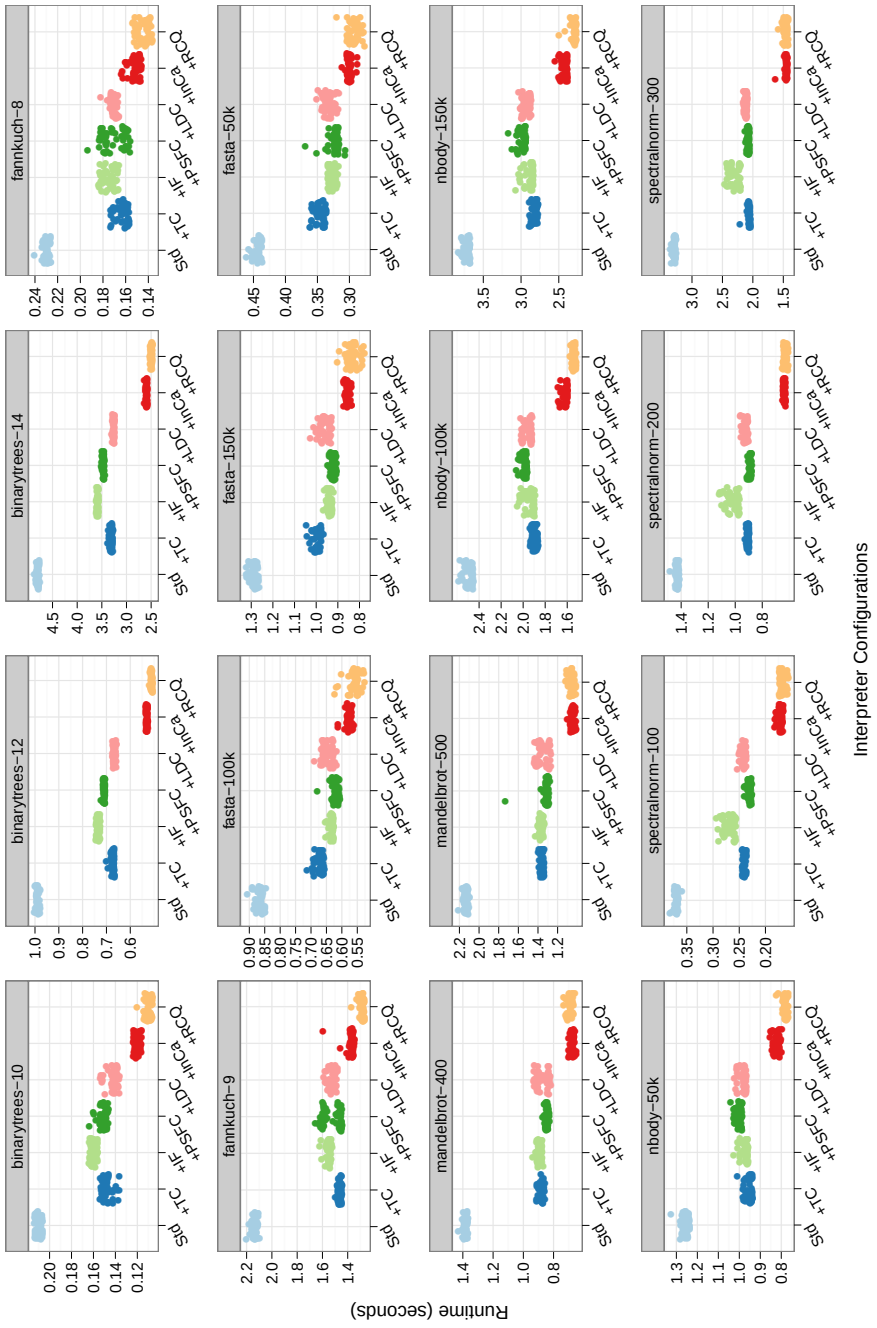


Figure B.1: Benchmark run-times per optimization technique.

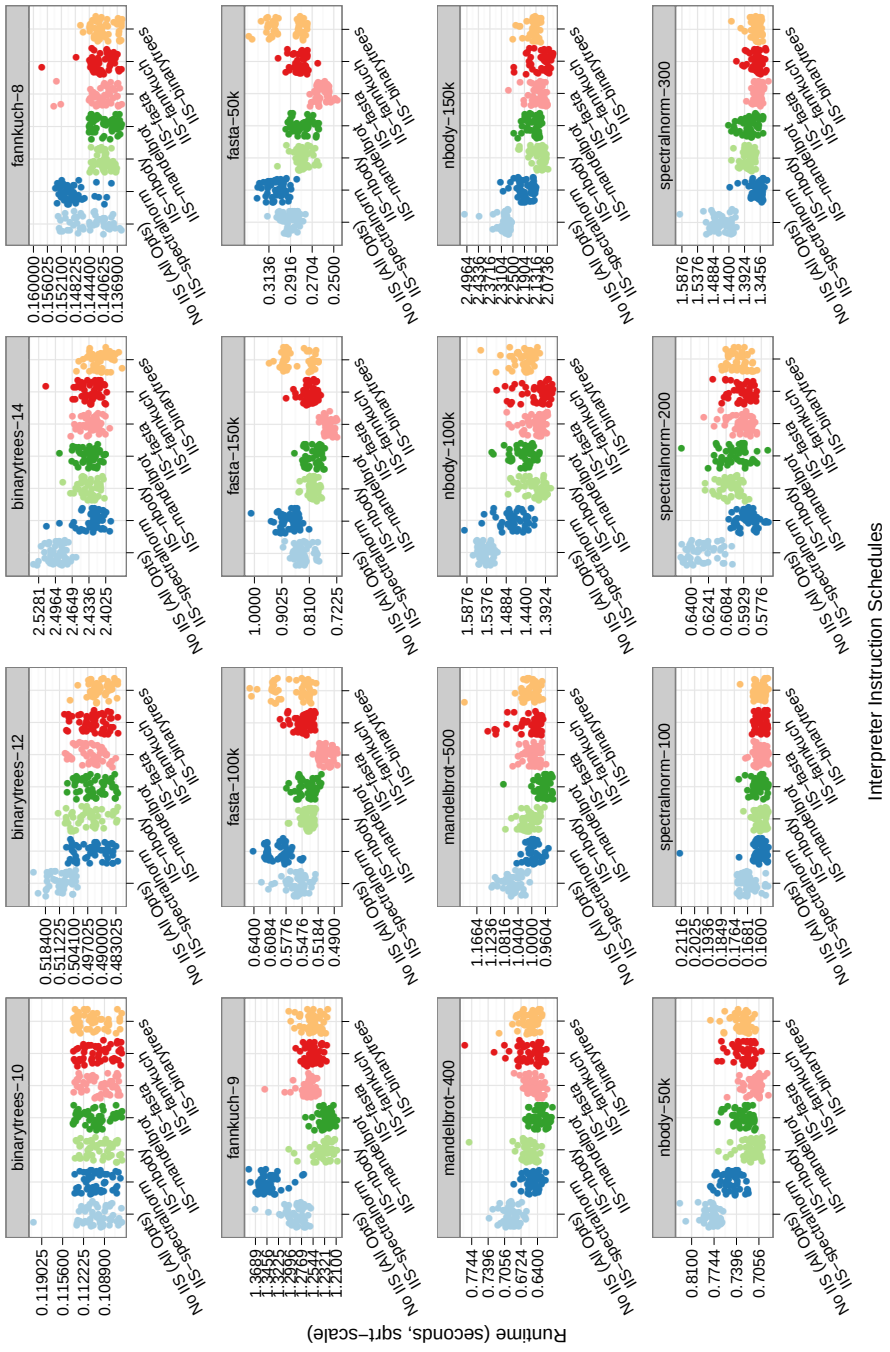


Figure B.2: Benchmark run-times for different interpreter instruction schedules.

Benchmark Run-time Results for PowerPC 970

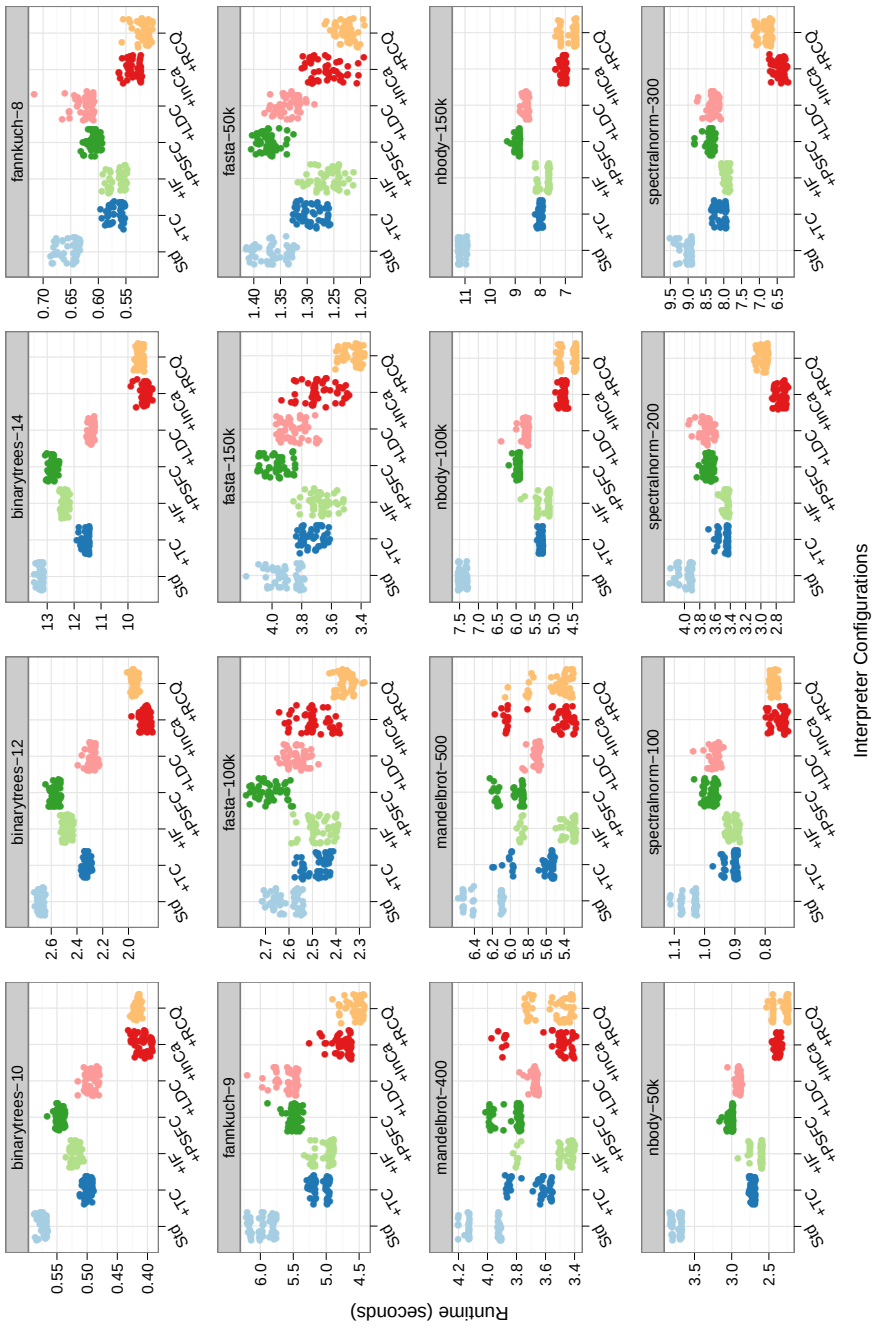


Figure B.3: Benchmark run-times per optimization technique.

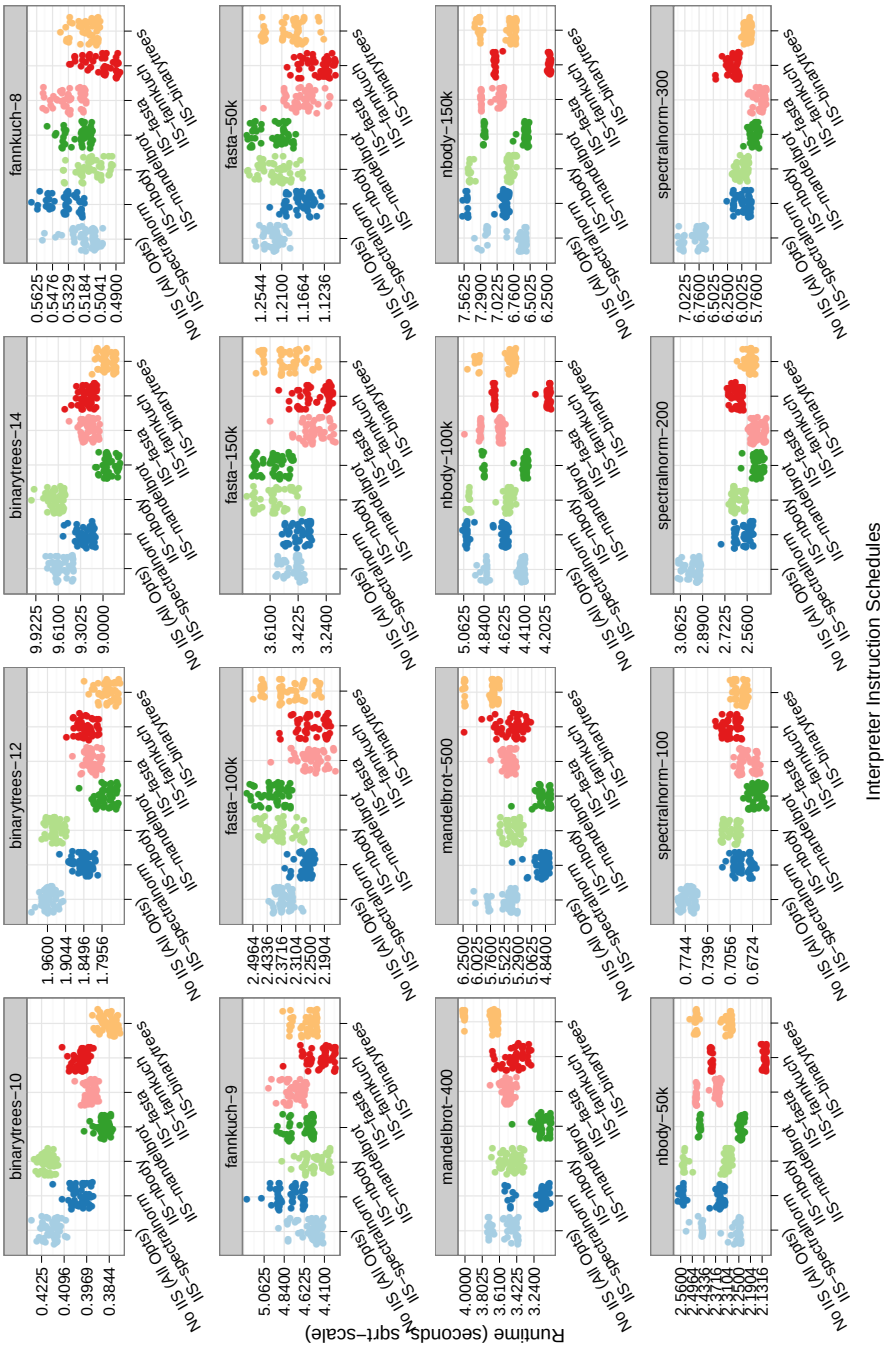


Figure B.4: Benchmark run-times for different interpreter instruction schedules.

Benchmark Run-time Results for Intel Atom N270

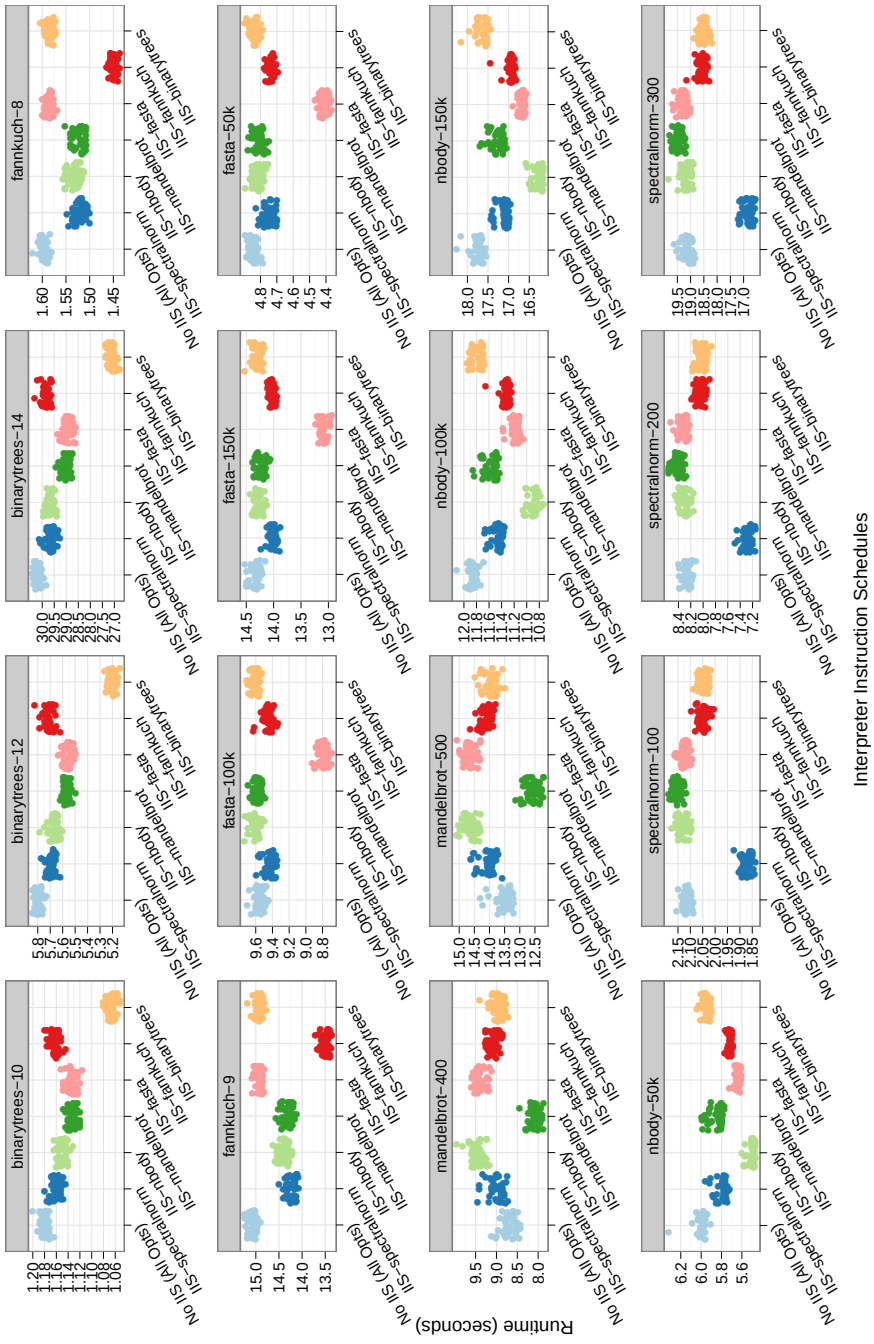


Figure B.5: Benchmark run-times for different interpreter instruction schedules.

Curriculum Vitae Stefan Brunthaler

Personal Data:

born	September 19th, 1980 in Ried/Innkreis, Austria.
raised	Braunau am Inn, Austria
languages	German, English, French (basic)
marital status	to be married to Mag. Catherine Bouvier in March 2011

Education:

2006 – *March 2011*	PhD student at the Compilers and Languages Group, Institute of Computer Languages, Vienna University of Technology. Advisor: Prof. Dr. Jens Knoop
2005 – 2006	PhD student at the Institute of System Software, University of Linz. Advisor: Prof. Dr. Hanspeter Mössenböck
2000 – 2004	Student of Software Engineering for Medical Purposes, University of Applied Sciences in Upper Austria, Hagenberg, Austria. Concluded diploma studies with a master's degree (*Dipl.-Ing. (FH)*).
1999	High school diploma (Matura), BG/BRG Braunau am Inn, Austria.

Research Interests:
Programming languages, compilation, interpretation, analysis, optimization, and verification; garbage collection; formal methods, logic, model checking; system software, browsers, debuggers, decompilers, and editors; legacy systems; domain specific languages.

Grants:

FWF 2010	Grant from the Austrian Research Council (FWF) for investigating speculative execution for the Python interpreter. Joint work with Prof. Dr. Jens Knoop and Prof. Dr. Anton Ertl. Project title: "S*py*culative", project volume: € 117,380.
ACM 2010	SIGPLAN PAC student grant for presenting at the Dynamic Languages Symposium (DLS '10) in Tahoe/Reno, Nevada, USA. Funded by SIGPLAN.
ACM 2010	Student Travel Award for presenting at the Symposium on Applied Computing (SAC '10) in Sierre, Switzerland. Funded by SIGAPP.

Invited Talks:

October 19, 2010	Mozilla Corporation, Mountain View, CA, USA.
October 15, 2010	Oracle/Sun Labs, Menlo Park, CA, USA.
October 11, 2010	University of California, Irvine, CA, USA.

Work Experience:

Feb 2009-Jul 2009	Software Architect at mms.ag GmbH, Vienna, Austria. *Projects*: Consulting architect on a mobile check-in solution for StarAlliance.
2006-Jan 2009	Software Architect at VeriSign AG, Vienna, Austria. *Projects*: Architect and technical lead for a mobile self-service platform.
Sep 2004-Dec 2005	Software Engineer/Scientific Employee at the Software Competence Center, Hagenberg, Austria. *Projects*: Re-engineering project of a big legacy application.
2004-2006	Software Engineer at 3united AG, Salzburg, Austria. *Projects*: Content distribution platform.
2003-2004	Software Engineer at Xidris GmbH, Salzburg, Austria. *Projects*: Sweep-stake application for Mobilkom Austria.
2000-2003	Software Engineer at Netzteil OEG, Braunau am Inn, Austria. (*defunct*) *Projects*: Several Projects for max.mobil (Austrian branch of T-Mobile)

Teaching Experience:

Legacy Systems	Part 3, Fall 2005, University of Applied Sciences, Hagenberg, Austria. (4h lecture + 4h lab) *Topics*: Part 3 of the series was about the genesis of legacy systems with focus on AS/400 as a hardware platform, and possible re-engineering scenarios.

Summer Schools:

LASER 2007	Summer School on Software Verification, Elba, Italy. Organized by: Prof. Dr. Bertrand Meyer and Prof. Dr. C.A.R. Hoare Swiss Federal Institute of Technology, Zürich (ETH).

Focus Courses:

Verification of Compilers	Summer	2008	Prof. Dr. Wolf Zimmermann
Computer Aided Verification	Summer	2008	Prof. Dr. Helmut Veith
Analysis and Verification	Winter	2007	Prof. Dr. Jens Knoop
Code Generation Techniques	Winter	2007	Prof. Dr. Andreas Krall
Optimizing Compilers	Winter	2006	Dr. Markus Schordan
Abstract Machines	Summer	2006	Prof. Dr. Andreas Krall
Seminar on Garbage Collection Techniques	Winter	2005	Prof. Dr. Hanspeter Mössenböck
System Software	Winter	2005	Prof. Dr. Hanspeter Mössenböck

Publications:

Peer reviewed:

C5. Stefan Brunthaler. Interpreter Instruction Scheduling. In *Proceedings of the 14th International Conference on Compiler Construction, Saarbrücken, Germany, March 26-April 3rd, 2011 (CC '11)*, *Lecture Notes in Computer Science*, Springer, 2011. To appear.

C4. Stefan Brunthaler. Efficient Interpretation using Quickening. In *Proceedings of the 6th Symposium on Dynamic Languages, Reno, Nevada, US, October 18, 2010 (DLS '10)*, pages 1–14, New York, NY, USA, 2010. ACM Press.

C3. Stefan Brunthaler. Inline Caching meets Quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25, 2010 (ECOOP '10)*, volume 6183/2010 of *Lecture Notes in Computer Science*, pages 429–451. Springer, 2010.

C2. Stefan Brunthaler. Efficient Inline Caching without Dynamic Translation. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, pages 2155–2156, New York, NY, USA, March 2010. ACM.

C1. Stefan Brunthaler. Virtual-Machine Abstraction and Optimization Techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09)*, pages 19–30, York, UK, March 2009. Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam, The Netherlands.

Technical reports as proceedings:

W2. Stefan Brunthaler. Inline Caching meets Quickening. In *Tagungsband des 15. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2009), (Maria Taferl, Österreich, 12.-14. Oktober 2009), Schriftenreihe des Instituts für Computersprachen, Technische Universität Wien, Bericht 2009-X-2 (2009)*, Ergänzungsband, pages 7-21.

W1. Stefan Brunthaler. Optimizing High Abstraction-Level Interpreters. In *Proceedings of the 26th Annual Workshop of the GI-FG 2.1.4 "Programmiersprachen und Rechenkonzepte" (Physikzentrum Bad Honnef, Germany, May 4-6, 2009), Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, Bericht Nr. 0915 (2009)*, pages 100-111.

Posters:

P2. Stefan Brunthaler and Jens Knoop. Elimination of Reference Count Operations in Bytecode Interpreters. In *Proceedings of the Junior Scientist Conference 2010 (JSC '10), Technische Universität Wien, Vienna, Austria, April 2010*, pages 39–40.

P1. Stefan Brunthaler and Jens Knoop. Optimizing the Python interpreter: Identifying performance bottlenecks. In *Proceedings of the Junior Scientist Conference 2008 (JSC '08), Technische Universität Wien, Vienna, Austria, November 2008*, pages 41–42.

Thesis:

T1. Stefan Brunthaler. Visualization and Management of Software Aspects. Master's thesis, University of Applied Sciences of Upper Austria, Hagenberg, Austria (2004).