

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

Master's Thesis

IMPROVING DYNAMIC LANGUAGE PERFORMANCE
USING JUST IN TIME COMPILATION

J.D. van Houten

Supervisor and tutor:
prof.dr. M.G.J. van den Brand

August 2011

Abstract

Dynamic languages are becoming increasingly popular in today's world. Javascript has become the ubiquitous language for the web browser, and Python and Ruby have become enormously popular on the server side. As a result efficient execution of these highly dynamic languages becomes paramount. JIT-compilers have been written for each of these languages but with mixed results. We present a simple approach to JIT compilation for dynamic languages (using a trivial dynamic language as working example) to illustrate where the largest performance bottlenecks lie and which optimizations have the most significant benefit. The thesis shows that even straightforward JIT compilation can lead to very efficient programs if some basic properties of a program can be deduced.

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Research questions	1
1.3	Motivation	2
2	Preliminaries	3
2.1	Concepts	3
2.2	Virtual Machine	3
2.3	JIT-Compilation	3
2.4	String Interning	3
2.5	Bytecode-Compiler	3
2.6	Opcode Dispatch	4
2.7	Virtual machine constraints	4
2.8	Class/Instance	4
2.9	Functions/Methods	4
2.10	Virtual Function	5
2.11	Messages	5
2.12	Static Single Assignment (SSA)	5
3	Performance Observations in Dynamic Languages	6
3.1	Introduction	6
3.2	Languages	6
3.3	Analysis	6
3.3.1	The Python Virtual Machine	7
3.3.2	The Ruby Virtual Machine	7
3.3.3	The Lua Virtual Machine	7
3.3.4	The OCaml Virtual Machine	8
3.4	Observations	8
3.5	Conclusion	9
4	Known Approaches to JIT Compiler Design	10
4.1	Overview	10
4.1.1	Incremental JIT compilation	10
4.1.2	Hotspot JIT compilation	10
4.1.3	Inference Based JIT compilation	11
4.1.4	Trace Based JIT compilation	11
4.2	Related Work	12
4.2.1	Localized Type Inference of Atomic Types in Python	12
4.2.2	Static Type Inference for Ruby	12
4.2.3	Psycho JIT Compiler	13
4.2.4	Starkiller	13
4.2.5	TraceMonkey	13
4.3	Conclusion	14

5	Conceptual Prototype JIT-Compiler	15
5.1	Introduction	15
5.2	Algorithm in C	15
5.3	Algorithm in C with function lookup	16
5.4	Algorithm via Bytecode Interpreter	17
5.4.1	Instruction Set	17
5.4.2	State space	17
5.4.3	The algorithm	18
5.4.4	Performance	18
5.5	Naive Compiler	20
5.6	Implementation	20
5.7	Performance	21
5.8	Conclusion	24
6	Extending the Prototype Compiler	25
6.1	Introduction	25
6.2	Language Design Preliminaries	25
6.3	Extension: Efficient Integer Math	27
6.4	Eliminating type checking	28
6.4.1	Type assertion elimination via type inference	29
6.4.2	Type assertion elimination via execution traces	30
6.4.3	Type assertion elimination via static analysis	31
6.5	Eliminating integer encoding and decoding	32
6.6	Eliminating overflow checks	32
6.7	Conclusion Integer Math	33
7	Extension: Efficient functions and methods	35
7.1	Optimizing Function Calls	36
7.1.1	Solution 1: direct function call	36
7.1.2	Solution 2: indirect function call	37
7.1.3	Solution 3: <i>vtable</i> based virtual function calls	37
7.1.4	Solution 4: self-patching direct function call	38
7.2	Optimizing Method Invocations	39
7.2.1	Caching	40
7.3	Conclusion Functions and Methods	41
8	Design Final Prototype Language and Compiler	42
8.1	Design Decisions	42
8.2	Garbage collection	43
8.2.1	Reference Counting	43
8.2.2	Mark-Sweep Garbage Collection	43
8.2.3	Mark-Compact Garbage Collection	44
8.2.4	Copying Garbage Collection	44
8.2.5	Generational Copying Garbage Collection	44
8.2.6	Conclusion Garbage Collection	45
8.3	Code Generation	45
8.3.1	TCC	45

8.3.2	LLVM	46
8.3.3	NASM, GAS	46
8.3.4	Libyasm	46
8.3.5	libjit	46
8.3.6	LuaJIT	47
8.4	DynASM	47
8.5	Register Usage	49
8.6	Pass by value	49
8.7	Local Variables	50
8.8	Conclusion	51
9	Conclusion	52
9.1	Results	52
9.2	Future Work	52
9.3	Reflection	53
9.4	Acknowledgments	53

List of Figures

1	Memory representation of object instance	25
2	Types of function calls	36
3	Stack and Type Stack side-by-side	50

List of Tables

1	Language complexity measured in lines of code	7
2	Performance Results	24
3	Performance Results	32

Listings

1	Algorithm in C	15
2	Algorithm in C with function lookup	16
3	Stack based Virtual Machine	19
4	Naive Compiler	22
5	Case analysis in typical Virtual Machine	27
6	Introspection in Dynamic Languages	35
7	Recursive Method Search Algorithm	39
8	Inline caching of method address	40
9	Compiler with inline assembly	48
10	Compiler with compiled assembly	48

1 Introduction

1.1 Introduction

In the past decades a tremendous amount effort has been put into improvements of compiler technology in terms of intelligence (creating more efficient code) and correctness (finding more flaws at compile time, refusal to compile ambiguous code). However, since the appearance of the programming languages Python and Lua in the early 90ies this started to change. Dynamically typed and high level programming languages quickly gained traction in academia and the commercial sector.

The defining characteristic of dynamic programming languages is that the program code is evaluated bit by bit as the program is running. This offers incredible flexibility: classes and functions can be created on the go, existing functions can be redefined depending on program conditions and so on. This flexibility comes at a cost: the dynamic programs are generally an order of magnitude slower than their statically typed and less expressive counterparts. This is partially because the flexibility provided by dynamic languages have an innate cost and partially because the dynamic languages are interpreted by simple virtual machines.

Interestingly, many programs written in dynamic languages are not highly dynamic in nature, so a even a simple compiler ought be able outperform a mature virtual machine by quite a margin. This is a task suited for just-in-time compilation. A just-in-time compiler uses a combination of static optimization strategies and information taken from the program as it is being evaluated to generate (equivalent) optimized code with as little overhead as possible.

1.2 Research questions

Investigate the performance bottlenecks of dynamic languages and investigate how a JIT-compiler can reduce or eliminate these bottlenecks.

1. Measure where the performance bottlenecks exist in straightforward virtual machines.
2. Inspect the effectiveness of straightforward JIT-compilation.
3. Investigate how the performance of a straightforward JIT-compiler can be improved.
4. Determine whether the performance gains of the JIT-compiler are worth the effort and increased implementation complexity.

1.3 Motivation

The design and implementation of virtual machines has been exhaustively investigated. A virtual machine prototype for a language can be built in a matter of days, if not hours. These virtual machines may be good candidates for a Just-In-Time compiler. With the aid of a compiler such a programming language may become viable for domains formerly reserved for efficient compiled languages.

The steps needed to make this transition are not self-evident. The JIT-compiler

- will need to generate assembly code which in turn will have to be transformed into machine code,
- needs to sensibly compile the virtual machine bytecode into suitable assembly code, and
- the JIT-compiler has to make sensible optimizations such that it does not have the same bottlenecks the virtual machine used to have.

These steps all come with numerous challenges. This aims to clarify some of the design decisions that need to be made in order to construct a Just-In-Time compiler and it provides numerous performance comparisons between a naive virtual machines and a straightforward Just-In-Time compiler.

2 Preliminaries

2.1 Concepts

This section defines and describes the concepts that will play a role in later sections. This document assumes a basic knowledge of virtual machines and compilers. No extensive prior knowledge about JIT-compilation techniques is assumed.

2.2 Virtual Machine

If a software written in a programming language must run on different platforms or when the programming language must make runtime guarantees an abstraction layer is needed in between the programming language and the target platform. This abstraction layer is called a virtual machine.

2.3 JIT-Compilation

Conventional compilers operate ahead-of-time. First a program is written by a programmer, then the compiler transforms the program into an executable binary and this binary is then executed. A *Just-In-Time* (JIT) compiler operates alongside the program it compiles. Consequently a JIT compiler is very constrained in terms of the memory and time it may use. The compilation overhead has to be earned back by the optimizations performed, which leads to.

2.4 String Interning

A dynamic language deals predominantly with strings and string comparisons are ordinarily slow operations. String interning is a technique that allows string equality checks in $O(1)$ time. The runtime environment has a global dictionary for storage of all interned strings. When a new string is interned the runtime environment will check if it occurs in the dictionary already, and if so return a pointer to that string. Otherwise an immutable copy of the string is added to the dictionary. Because every interned string is unique in memory an equality test between interned strings can now be accomplished via comparison of the pointers of the interned strings. This technique is used by the Matz Ruby interpreter¹ and by the ATerm Library[7].

2.5 Bytecode-Compiler

A bytecode compiler takes an Abstract Syntax Tree (AST) as supplied by the parser and compiles it into a stream of bytecode instructions. The bytecode is typically a cross platform high level opcode. To illustrate: the expression "3 + 5 / 2" may be compiled into the bytecode `PUSH 3; PUSH 5; PUSH 2; DIV; ADD`; suitable for evaluation by a stack machine. A *fat bytecode* is a bytecode with a substantial amount of functionality. For instance an instruction that does a method lookup or an instruction that allocates and initializes a data structure are considered *fat*. An instruction is considered *lean* when it can be implemented with a few machine code instructions, e.g. the bytecode `PUSH`². Simple virtual machines

¹Interned strings are known as *Symbols* in Ruby parlance

²If the `PUSH` instruction has to check for available stack space and relocate if no space is available the stack it can no longer be considered *lean*.

tend to use fat high level bytecodes for simplicity and efficiency. Fat bytecodes are more efficient because of the reduced overhead in *opcode dispatch* and because the compiler of the virtual machine itself can more easily optimize *fat* instructions. On the other hand, when (JIT)-compiling into native code *lean* instructions can be optimized more efficiently.

2.6 Opcode Dispatch

A virtual machine executes a program by evaluating the bytecode emitted by the bytecode-compiler one by one. A simple virtual machine may use a switch statement or cascading if-else statements for evaluation of the bytecode instructions. A slightly more efficient virtual machine may dispatch the instructions via a dispatch table. The method of mapping an instruction to the virtual machine logic that evaluates the instruction is known as Opcode Dispatch. There is no equivalent of Opcode Dispatch for a JIT-compiled function; there is no need for it.

2.7 Virtual machine constraints

In a dynamic language the runtime environment must know at least the type of every variable in the program at any point in time. This is not possible in a language like C where the type information is lost during compilation time. There are always conflicting trade-offs between memory and CPU usage, between complexity and efficiency. A C compiler is allowed to apply drastic transformations to the source code to create faster executable, a JIT-compiler for a dynamic language may not.

Most user defined objects primarily serve to make the program code more readable than the purely procedural counterpart, to decrease coupling between components, to explicitly define an interface and to encapsulate the implementation details. These are all high level concerns and on this level of abstraction slight inefficiencies in the virtual machine are unimportant. However, modern dynamic languages follow the "Everything is an object" model from Smalltalk[14]. This means that programmers must be able to query any object, whether it is a high level user defined object or something as primitive as an integer literal.

2.8 Class/Instance

In a dynamic language a class is a type and a class has an interface. The term *Object* usually refers to an instance of a class. The class interface defines which methods are known to that class. This interface may not be fixed: methods may be added to and removed from a class at any time during execution of the program. A function-object is an object that contains a function so that it may be assigned to a variable, stored in a container, and so forth.

2.9 Functions/Methods

A function is a procedure that belongs to a namespace but is not part of a specific class hierarchy. A method is a procedure that belongs to a class. Class-methods are methods that belong to the class itself, ordinary methods belong to an instance of a class. Methods are polymorphic, functions are not.

2.10 Virtual Function

Statically compiled languages such as C++ implement polymorphism by means of virtual functions. A virtual function is a function that has multiple implementations in different classes in a class hierarchy. Which one of the implementations is invoked does not depend on the arguments of the function (that would be overloading) but on the type of the instance on which the function is invoked.

2.11 Messages

A method of an instance of a class is called by means of a *message*. A message, like a function, has a name and a number of arguments. When a message is dispatched to an object a special function is called. Typically, this function will check if a method exists with the same name as the message and if so invoke that method and throw an exception otherwise.

2.12 Static Single Assignment (SSA)

Single Static Assignment is a form of a program fragment where every variable is assigned exactly once. A program fragment can be transformed into SSA form by splitting variables into multiple versions, one for each assignment. A program fragment in SSA form simplifies optimization techniques such as *constant propagation*, *register allocation* and *dead code elimination*. It also simplifies type analysis since the type of every variable is immutable in SSA form.

3 Performance Observations in Dynamic Languages

3.1 Introduction

In this chapter the performance characteristics of four multiple virtual machines are discussed. Established benchmarks are used to give an indication of the relative performance of these virtual machines. The source code of the virtual machines is then inspected to determine which elements of the design are shared by efficient and inefficient virtual machines. By bringing the performance benchmarks and the differences in virtual machine design together we can distinguish between meaningful and inconsequential attributes of virtual machine design with regard to performance.

3.2 Languages

In order to get the most meaningful results two very similar virtual machines are picked which are, according to their respective authors, not written with performance as a primary goal. These are the virtual machines for Python and Ruby. Both languages are dynamic in nature and have very similar semantics. The virtual machines are therefore expected to be regular virtual machines with very similar designs, both on the abstract levels and down to the implementation details. Yet, according to the Alioth Language benchmarks Python outperforms Ruby on average by a factor of five. Note that these are mostly algorithmic and numerical benchmarks (it includes a mandelbrot and n-body simulation) and are therefore not particularly representative of the average Python or Ruby program. Nonetheless, Python consistently outperforms Ruby by a very significant margin and given that CPU-bound programs are atypical in the first place the benchmarks do demonstrate there is a real difference in the virtual machines themselves.

The third virtual machine covered is Lua. Lua is also a flexible dynamically typed language, mostly used for embedded software and as a module embedded in a statically compiled language such as C++. In contrast to the design of Python and Ruby Lua has been designed from the ground up with performance characteristics and a small memory footprint in mind[13]. According to the same benchmarks cited above Lua outperforms Ruby on average by a factor of nine over 9 numeric and algorithmic benchmarks and Python by less than a factor of two.

The fourth and final virtual machine is OCaml. Since OCaml is in the ML family of languages much more information is available to the runtime environment. OCaml also has a compiler, which generates efficient code that reaches at least 50% of the performance of the equivalent C++ program³. OCaml is unusual in the sense that it has both a virtual machine and a compiler even though it is statically typed.

3.3 Analysis

The analysis of Lua is relatively straightforward since it has an explicit document outlining the design philosophy and the motivations thereof. For the other languages observations are based on the implementation (source code) of the virtual machines themselves and where applicable, the documentation.

Although Python, Lua and to a lesser extent Ruby all value simplicity and straightforward implementation the difference in code size of the respective language environments is immense.

³According to Xavier Leroy, the primary developer of the OCaml compiler

	kLoc VM	kLoc Std Libraries - C	kLoc Std Libraries
Python 2.4	166	137	202
Ruby 1.8	65	68	98
Lua 5.1	7	3	0
OCaml 3.1	15	15	137

Table 1: Language complexity measured in lines of code

Table 1 shows the difference in language size as measured in lines of code, excluding empty lines and comments⁴. We acknowledge that "lines of code" is not an accurate metric of project complexity, but it can be indicative of complexity.

In the table the first column refers to the lines of C code of the virtual machine. The second column refers to the number of lines of C code of the core libraries (basic data structures and algorithms). The third column refers to the library code written in the language itself. The unit tests, where applicable, do not count toward any of the totals.

3.3.1 The Python Virtual Machine

Python is a classical stack based virtual machine. A simple compiler turns the AST representation of the source code into bytecode. Unreachable objects are automatically freed using reference counting.

As shown in table 1 Python is the biggest language by far. Python's slogan is "Batteries Included" and it shows. Additionally, most of the language is implemented in C for performance reasons.

3.3.2 The Ruby Virtual Machine

Like Python, Ruby is a stack based virtual machine. It uses mark and sweep garbage collection. The virtual machine does not cleanly distinguish between the parser, abstract syntax tree and the compiler. The abstract syntax tree undergoes a number of transformations and is then directly evaluated. This was probably a mistake and a virtual machine for Ruby is under development that uses a bytecode layer for performance and cleaner separation of concerns.

3.3.3 The Lua Virtual Machine

Lua 5.0 is tiny compared to the previous two virtual machines. There is only one numeric type (double-precision floating point) and only a 35 different instructions. The virtual machine is also designed with embedability and portability in mind, and this is evident from its design. Since two virtual machines have already been covered in detail only meaningful deviations from the standard approach will be mentioned here.

Lua is noteworthy because of its minimalist approach. There is only a single numeric type (a double-precision floating point), there are only 35 instructions and there is a single data structure named a *table* (dictionary). Also unlike the other virtual machines Lua is a register machine. Earlier versions of Lua were a stack machines, and the transition to a register based approach was motivated primary by performance[13].

⁴Data was generated using David A. Wheeler's 'SLOCCount'.

3.3.4 The OCaml Virtual Machine

OCaml is a statically typed language with type inference that belongs to the ML family of languages. It consists of an interpreter and an optimizing compiler which generates code that is competitive with C in terms of performance and memory usage⁵. The virtual machine consists of only 15000 lines of C code: the bare minimum needed to evaluate the program that produces the optimizing compiler (written in ML), which is then executed to create an optimized version of itself. The OCaml compiler cleanly distinguishes between the different modules: parser, compiler, interpreter, code generator.

3.4 Observations

Based on the source code and design documents of the programming languages the following observations are made:

- **Type Checking.** The bytecode compilers of Python, Ruby and Lua can make very few optimizations based on type because the type information is not explicitly stated in the source code and the virtual machines make no attempt to guess or derive what the types of expressions will be. Consequently, every operation on an object requires a number of runtime checks. Especially for basic numerical algorithms verifying whether the objects have the expected types is operationally much more difficult than the mathematical operation itself.
- **Opcode Dispatch.** It turns out that the choice of opcodes and the granularity of opcodes has a meaningful influence on the design of a virtual machine. The virtual machines have few high-level (fat) opcodes to allow for simpler and more efficient dispatch mechanisms. Additionally, a program will have a shorter bytecode listing so fewer dispatches are necessary during evaluation of the program. The opposite approach of lean opcodes (that contain less functionality) can be combined into less redundant and therefore more efficient code. The virtual machines all favor a fat-opcode approach which indicates that either the overhead of opcode dispatch itself is still very significant (the extra effort required to dispatch many lean opcodes outweighs their benefit) or the complexity of lean-opcode virtual machines is (deemed) to outweigh the benefits.
- **Dictionary access.** Dictionary lookups are frequent and lookup results are not cached. Given that function and method addresses very rarely change there may be room for improvement here. Python accesses local variables via an ordinary array instead of via a dictionary. This optimization confirms that dictionary lookups can indeed be a significant factor in performance. Method calls seem especially expensive, given that the chain of parent classes has to be traversed in order to locate the method in question.
- **Numeric operations.** Ruby and Python globally use arbitrary precision integers. To partially mitigate this problem both virtual machines repeatedly check whether an integer is within a certain small integer bound, and if so, the integer is treated as such. Otherwise, the integer is quietly promoted and the less efficient (but mathematically sound) arbitrary integer implementation is used instead. Here too the observation holds that edge case where an integer needs to be promoted occurs only very rarely yet (and

⁵According to the Alioth Language Shootout

often can never occur) yet the assertions are made multiple times per statement. The frequency with which numerical operations occur in any program imply even small improvements may be worthwhile. Lua has only one numeric type: double precision floating point. This makes the virtual machine significantly less complex and no case analysis is needed within an opcode to distinguish between the numeric types of the operands.

- VM complexity. The Python, Ruby and OCaml environments comprise over hundreds of thousands of lines of code. Python and Ruby deliberately started out as simple languages but they have grown over the years into the large projects they are today. OCaml has grown in complexity also, but because most of the OCaml compiler and standard library is written in OCaml (which is more expressive and less verbose than C) the complexity is relatively low. The virtual machine has an excellent straightforward implementation because it does not need to break any performance records. Lua of course has the simplest implementation of all.

3.5 Conclusion

Given the observations described above we conclude that a number of characteristics in popular virtual machines are likely to have a significant adverse affect on performance. However, much of the overhead is inherent to the virtual machine approach. Register and stack based machines suffer from the dispatch overhead and excessive runtime assertions. OCaml is significantly faster and eliminates both problems but it can only do so because of the strict type inference. A just-in-time compiler can make all sorts of optimizations contextually, and can also (trivially) eliminate opcode dispatch overhead and other virtual machine inefficiencies.

The next chapter covers just-in-time compilation techniques and which of those techniques look most promising as a solution to the aforementioned problems.

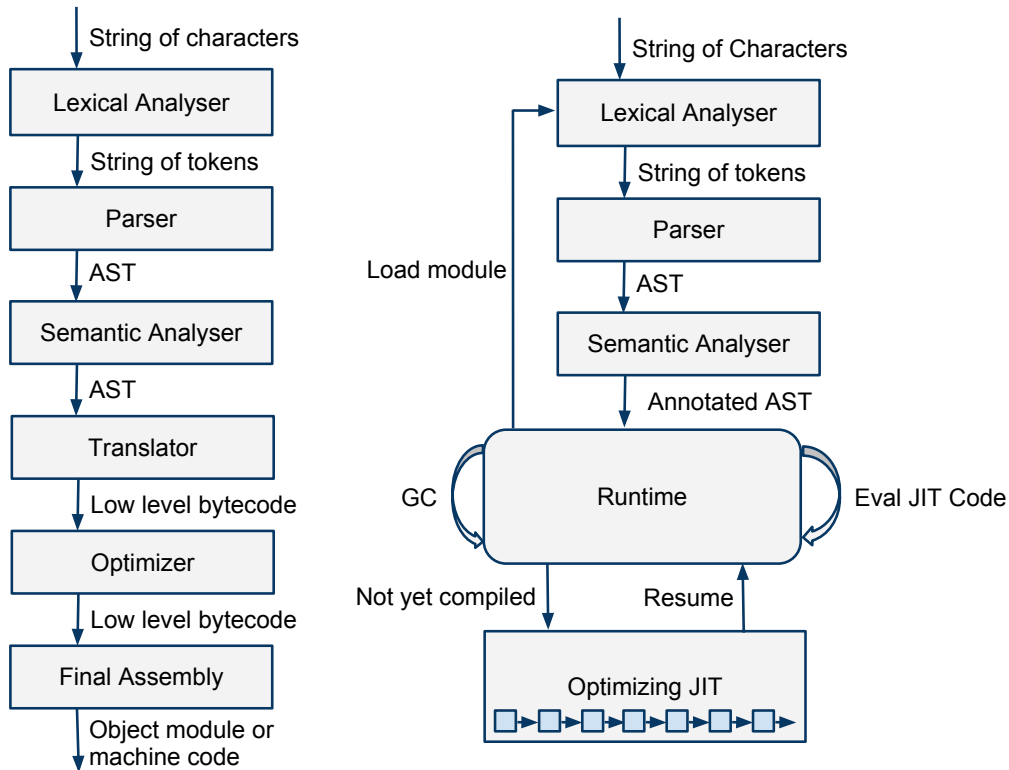


Figure 1:
Optimizing Compiler Design vs Incremental Optimizing JIT Compiler

4 Known Approaches to JIT Compiler Design

4.1 Overview

The purpose of a just-in-time compiler is to improve the performance or memory usage of a program. Like a conventional *ahead-of-time* compiler a JIT compiler transforms the Abstract Syntax Tree (AST) of the program into machine code. When this compilation phase happens depends on the type of JIT compiler.

- incremental JIT compilation
- hotspot based JIT compilation
- inference based JIT compilation
- tracing JIT compilation

4.1.1 Incremental JIT compilation

The first compilation strategy is incremental JIT compilation. The first time a function or method is called the JIT-compiler will interrupt the execution of the program and JIT-compile the a function or method in question. Because the compilation happens *Just-In-Time* some optimizations become possible. For instance, function can be JIT-compiled multiple

times to specialize on the function parameters or a function can be compiled with other assumptions in mind, such as "all integers are small" or "objects are non-null". If one of these assumptions is violated (this has to be verified at runtime by means of assertions) the program will transparently fall back to a less efficient version of the same function that makes no assumptions about variables and types.

This compilation strategy has a number of desirable characteristics. It is very efficient and it can completely replace a bytecode interpreter. The implementation is also reasonably straightforward. In later chapters we will show that the implementation of a simple incremental JIT compiler is not substantially more difficult than the implementation of a bytecode interpreter for the same language.

4.1.2 Hotspot JIT compilation

The second compilation strategy is Hotspot based JIT compilation, first used by the Java Hotspot compiler. Initially all code is interpreted by a virtual machine (cold code). The virtual machine keeps track of system resources used by the user program (memory/cpu). The functions that turn out to be the most resource intensive are then targeted for JIT-compilation. This approach is more complex than straightforward Incremental JIT Compilation because in addition to a JIT compiler a semantically equivalent virtual machine is needed. The JIT compiler can afford to spend more time on optimization of a function when it is known that the function is a performance bottleneck. In practice it is difficult to discover good heuristics that determine whether functions are worthwhile targets for optimization. Since functions are individually targeted for optimization a HotSpot compiler has no more context information than an Incremental JIT compiler.

How effective a JIT compiler can be when optimizing at the function level depends on the information the programming language makes available to the compiler. For instance, if the programming language is (mostly) statically typed and if classes and types cannot be extended or manipulated at runtime meaningful performance gains can be made: method calls can be resolved ahead of time, primitives can be unboxed and so on. Early versions of the Java HotSpot compiler achieved significant performance improvements in this manner. On the other hand, if a programming is highly dynamic the possibilities for optimization are very limited. The JIT-compiler can make assumptions about the types of expressions and make optimizations based on those assumptions.

4.1.3 Inference Based JIT compilation

The lack of exact type information is a significant problem for efficient JIT compilation. For instance, mathematical operations cannot be optimized if the compiler is not aware which expressions have numeric types.

A method is JIT compiled the first time it is invoked. At that time some exact type information is available: the types of the function arguments, the types of all global variables, types of all member variables of the class and the types of all literals in the body of the method. An inference based JIT compiler uses the available type information to predict the types of expressions in the body of the method. These type assumptions are then used as a basis for further performance optimizations. Type assertions are inserted into the generated code for each assumption made by the inference engine. When a type assertion fails during the evaluation of the optimized code the JIT-compiler may attempt to compile the method again

given this new type information. Alternatively, execution may continue with the original unoptimized code. A heuristic is used to determine which of these two options is taken.

The underlying assumption is that as long as variables in a dynamic program are relatively *stable* and that straightforward type inference therefore yields mostly correct type assumptions. Practical issues, implementation and results are described in Chapter 10.

4.1.4 Trace Based JIT compilation

In a dynamic language every variable references an object and interactions with objects happen via messages. So even the most simple and straightforward program statements touch multiple methods. This makes optimizations on the function or method level ineffective. A *Trace Based JIT Compiler* optimizes a code path that may span multiple functions and methods. By carrying the knowledge about variables between statements on the code path a tracing JIT-compiler can be much more effective[9].

A Trace Based compiler goes through multiple stages:

1. incremental bytecode or JIT compilation,
2. tracing a hot code path, and
3. JIT-compiling the trace using the collected information.

A suitable start point (also known as a trace anchor) for a program trace is the start of a hot loop construct. This can be accomplished by adding a counter before every upward jump in the generated code. If a jump counter exceeds a predetermined number the corresponding loop is considered hot and the tracing process will commence. The end of a trace is simply the end of the loop, i.e. the instruction following the upward jump.

The JIT compiler will now recompile the function such that trace information is collected at every step. For every assignment the type of the variable is recorded. For every method invocation the address of the method is recorded. Since only a single iteration of the loop is needed to collect the trace information the JIT-compiler can greedily collect any information it might need. Methods that are invoked during the tracing phase are also recompiled for trace recording. This collection and recompilation process continues recursively until a single iteration of the original loop has been completed. Branches that occur on the trace path are replaced by guards in the optimized code. When a guard fails the trace ends and execution continues on the original (less optimized) code.

A single-entry multiple-exit *trace tree*[12] can be constructed that connects multiple traces at branch points in the code. This allows branches to occur within hot code paths without leaving trace-optimized paths. When a program contains a nested loop the most naïve approach will not work well. The inner loop will be trace-optimized first. The outer loop will then be considered for trace optimization but a single iteration of the outer loop may require evaluation of the inner loop. The inner loop may have a complex trace tree and for every path through the trace tree a new trace has to be created in the trace tree of the outer loop. This is a form of unintended tail duplication[12]. The *TraceMonkey Javascript JIT Compiler* solves this problem by recording nested trace trees. When an inner loop is located the compiler will search for an existing trace tree for that loop and call that trace tree instead.

4.2 Related Work

This section covers related work and alternative approaches to improve the performance and correctness of dynamic programming languages.

4.2.1 Localized Type Inference of Atomic Types in Python

B. Cannon[8] investigated to the implications of adding type inference to Python of atomic (primitive) types. The idea is to add type-specific bytecodes to Python. These type-specific bytecodes do not use any type checks during runtime. A type inference algorithm is used to determine where these type-specific bytecodes can be used instead of the existing type-agnostic ones.

Although the type inference algorithm is able to detect many places where type-safe bytecodes can be used, no significant performance benefit has been achieved. The author concludes that a 1% performance boost is not worth the additional complexity to the Python compiler.

The meager performance boost may be explained by the significant amount of bookkeeping involved with every Python bytecode. The runtime environment has to do a long jump to the opcode functionality, check for null references, check for over and underflows for integer math, and dictionary lookups are needed for every function call or member variable access. These are expensive operations compared to a type assertion that consists of a pointer comparison.

4.2.2 Static Type Inference for Ruby

In the paper "Static Type Inference for Ruby"[11] Furr et al investigate whether Ruby programs can be statically type-checked to provide early errors and to aid documentation (by providing typed function signatures). A tool *DRuby* is created based on static type inference, aided by type annotations where necessary to verify the type-safety of Ruby programs. These type annotations are checked at run-time, so the combination of the static and run-time checks deliver strict type guarantees. Because of the highly dynamic nature of Ruby, this system is not flawless: both false positives and false negatives happen on occasion.

The paper concludes that many Ruby programs can be type-inferred and that by doing so many of the advantages of statically typed languages can be brought to dynamic languages.

4.2.3 Psycho JIT Compiler

The Psycho just-in-time compiler for Python infers from the values your program manipulates some restrictions about the variables. For example, variables always containing an integer value, or a list of strings of length 1, or a tuple whose first item is zero. Using these constraints, more efficient machine code can be emitted.

Psycho's biggest advantage is that it can be used with existing Python code without any modification. A single function call is sufficient to tell Psycho which functions to optimize. If the just-in-time specialized code delivers a significant performance improvement the programmer's goal has been reached. If, on the other hand, the performance improvement is insufficient the performance-critical code can then be written as a C module. Psycho excels at improving the performance of numerical algorithms (where 100x improvements are possible), but is mostly ineffective at optimizing more general purpose Python code. Because Psycho generates multiple specializations at runtime, based on the types of the variables in

the program fragment, the just-in-time compiler has to use a significant amount of memory. A second (minor) downside is that because Psycho optimizes locally no optimization attempts can be made across program fragments.

4.2.4 Starkiller

The Starkiller[19] project ("A static Type Inferencer and Compiler for Python") converts Python programs to semantically equivalent C++ programs, using type inference (Agesen's Cartesian Product Algorithm). The motivation for the Starkiller project is threefold: (1) to bring the performance of Python closer to the level of C++ and (2) to statically verify type correctness in Python programs, to aid debugging and reliability of Python software and (3) to make Python more accessible for programmers who are not used to dynamic languages.

The author M. Salib observes that without comprehensive Type Inference no substantial performance gains can be made (the other work covered in the previous sections confirms this). In order for this approach to work Python programs must adhere to a number of constraints. The use of `eval` is prohibited, as is the dynamic redefinition of class types. The Starkiller compiler also deviates from Python's language specification in some respects. Register-sized integers in Python are silently promoted to arbitrary-precision integers on overflow, but Starkiller asks the programmer to pick between the two types of integers.

Starkiller reports substantial performance improvements for micro-benchmark of Fibonacci and factorial computations, but no benchmarks have been documented for more complex programs. M. Salib mentions that benchmarks like this are "as favorable to Starkiller as possible".

4.2.5 TraceMonkey

The TraceMonkey[12] JIT compiler for Javascript takes a trace-based approach to optimization of highly dynamic code. Frequently executed loop traces are identified at runtime. These *hot* traces are then specialized for the actual dynamic types recorded during the first evaluation of the loop body. Trace trees of different specializations are constructed to maximize the time spent in optimized machine code.

TraceMonkey improves the performance of some standard Javascript benchmarks by 10x compared to the baseline interpreter SpiderMonkey[18]. Compared to the modern optimizing Javascript compilers SquirrelFish Extreme[24] and Chrome V8[5] the TraceMonkey compiler outperforms the others in 9 of 26 benchmarks compared to the baseline interpreter. Programs that are amenable to tracing have reached speedups of 2x to 20x. The trace recording and compilation phases currently take considerable processing time and the authors believe that substantial improvements can be made in that area.

4.3 Conclusion

There are at least four different approaches to JIT-compilation for dynamic languages: incremental JIT compilation, inference based JIT compilation, hotspot based JIT compilation and trace based JIT compilation.

Trace based JIT compilation strategies, such as those used by TraceMonkey, can lead to very good results. Unfortunately, it is the most complex of the compilation strategies. For trace based compilation the runtime environment has to have an efficient interpreter, the

capability to record traces and the ability to seamlessly jump between the trace-optimized code and the virtual machine. This kind complexity is generally undesirable.

A Hotspot based JIT compiler also requires a virtual machine for the ordinary evaluation of code and it depends to a large extent on good heuristics that determine which methods deserve optimization and which do not. This is more suited for a statically typed language such as Java with long running processes (so that it may eventually become efficient) than for a very dynamic language without readily available type information.

This leaves incremental- and inference-based approaches to JIT compilation.

5 Conceptual Prototype JIT-Compiler

5.1 Introduction

In order to build a JIT-compiler that generates efficient code a performance baseline is useful. In other words: what kind of performance can be expected from a trivial compiler that naively translates bytecode into machine code? We should expect that a naive compiler yields significantly faster code than the equivalent virtual machine counterpart. This is what this chapter sets out to determine.

First a trivial CPU-bound algorithm is chosen as the guiding benchmark, in this case the canonical recursive Fibonacci function. It goes without saying that a Fibonacci calculation is not representative of programs written in dynamic languages. The program is very short, highly recursive, mathematical in nature and deliberately inefficient. Typical programs are none of these things. However, in order for a recursive Fibonacci function to execute efficiently the language has to have efficient integer math, efficient method calls and no other substantial overhead, which are *necessary* but not *sufficient* conditions for high performance.

First the algorithm is implemented in C. The C version will set the upper bound on the expected performance. Subsequently the algorithm is executed on a number of virtual machines (Ruby, Python, Lua), to determine the lower bound on the expected performance. A lower bound is needed because a JIT-compiler must at least outperform bytecode based virtual machines and naive interpreters.

Although these lower and upper bounds are reasonable, they do not truly represent the difference in language efficiency since the dynamic languages do substantially more (useful) work. In order to compensate for this effect a trivial virtual machine will be created that is functionally equivalent to the compiler, but has the inefficiencies of opcode dispatch and a bytecode layer. This should indicate to which parts of a virtual machine the execution time can be attributed.

5.2 Algorithm in C

Listing 1: Algorithm in C

```
1  int fib(int n) {
2      if (n < 2) return n;
3      return fib(n - 1) + fib(n - 2);
4  }
5
6  int main(int argc, char** argv) {
7      int s = 0;
8      for (int i = 0; i < 30; i++) {
9          s += fib(i);
10     }
11     printf("%d\n", s);
12 }
```

The C version of the Fibonacci sum program takes 18 milliseconds to execute. The 18ms needed by C gives us an upper bound on the efficiency that can be (realistically) attained with a JIT-compiler. For comparison, CPython takes over 1.5 seconds and Ruby needs over 6 seconds for the same algorithm. So Ruby is over two hundred times slower. See table 3 at the end of the chapter for details.

5.3 Algorithm in C with function lookup

Listing 2: Algorithm in C with function lookup

```
std::map<std::string, void*> g_functions;
2 typedef int (*fnptr)(int);

4 void* get_function(const char* name) {
    std::map<std::string, void*>::const_iterator it;
6     it = g_functions.find(std::string(name));
    if (it != g_functions.end()) return it->second;
8     else return NULL;
}

10
11 int fib(int n) {
12     if (n < 2) return n;
    fnptr p1 = (fnptr) get_function("fib");
14     fnptr p2 = (fnptr) get_function("fib");
    if (p1 && p2) return p1(n - 1) + p2(n - 2);
16     else throw std::string("No_such_function:_fib");
}

18
19 int main(int argc, char** argv) {
20     g_functions.insert( std::make_pair("fib", (void*) fib));
    int s = 0;
22     for (int i = 0; i < 30; i++) {
        fnptr p = (fnptr) get_function("fib");
24         if (p) s += p(i);
        else throw std::string("No_such_function:_fib");
26     }
    printf("%d\n", s);
28 }
```

Dynamic languages call functions by name via a dictionary lookup. Since functions and methods can be redefined at runtime virtual machines generally have to retrieve the address of the function every time it is called. This process can be simulated in C by storing the function pointer in a dictionary. The `std::map` dictionary is part of the C++ Standard Library and is implemented with a red-black tree. The dictionary is balanced and stable⁶, and has $O(\lg(N))$ lookup cost.

By dynamically retrieving the Fibonacci function in the inner loop the computation time increases significantly: it now takes 950 milliseconds for the answer to return. The `std::map` declared on line 1 is a dictionary that maps strings to function pointers. It is noteworthy that these dictionary lookups represent at least 900ms out of the 1.5 seconds Python needs to run the algorithm. The remaining 600ms are expected to be overhead in opcode dispatch, arbitrary precision integer arithmetic, and other bookkeeping. In the next section a trivial bytecode based virtual machine is used to test this hypothesis.

⁶Stable: dictionary items are not moved in memory when mutating the dictionary: a pointer to an item in the dictionary remains valid until the item is removed from the dictionary.

5.4 Algorithm via Bytecode Interpreter

A stack based virtual machine consists of: a set of instructions, a stack and a number of registers. A virtual machine that is capable of executing the recursive Fibonacci algorithm is given in this section. Emphasis lies on simplicity: the virtual machine is only 20 lines of C++ code long.

5.4.1 Instruction Set

Instructions may have up to one argument. The `ADD` uses the accumulator and the top of the stack, whereas `ADD-C` uses the accumulator and an embedded integer. Instructions with an embedded argument end by convention with a letter that denotes the type of the argument. The postfix `S` denotes a string constant and `C` an constant integer. The following instructions are defined:

<code>POP</code>	Takes an argument from the stack and places it into the accumulator.
<code>PUSH</code>	Push the value of the accumulator onto the stack.
<code>SWAP</code>	Swap the topmost and second topmost values on the stack.
<code>LT-C x</code>	Unless the accumulator is smaller than the given constant skip the next instruction.
<code>INT-C x</code>	Assign x to the accumulator.
<code>ADD</code>	Pop a value from the stack and add it to the accumulator.
<code>ADD-C x</code>	Add the given constant to the accumulator.
<code>CALL-S x</code>	Lookup the function by identified by the given constant. Run it.
<code>RET</code>	Return from the function.

5.4.2 State space

The virtual machine has a single register: the *accumulator*. The accumulator is used as an implicit argument to every instruction. See the sidebar for the importance of an accumulator register. The following notation is used to represent the state of the stack (topmost element on the left):

$$[1 | 2 | 3 | \dots]$$

The stack, accumulator, instruction pointer and the call stack (implicit) comprise the entire statespace of the virtual machine.

5.4.3 The algorithm

The recursive Fibonacci function can be implemented with 8 instructions described above in the following manner:

LT-C 2; RET	Return if the accumulator (the argument) is less than 2.
PUSH	Push the accumulator onto the stack [n ...].
ADD-C -1	Subtract 1 from the accumulator.
CALL-S "fib"; PUSH	Call $fib(n - 1)$ and push the result onto the stack.
SWAP	Stack [$fib(n - 1)$ n n ...] becomes [n $fib(n - 1)$ n ...].
POP; ADD-C -2	Pop n from the stack and assign $n - 2$ to the accumulator.
CALL-S "fib"	Call $fib(n - 2)$, the stack becomes [$fib(n - 1)$ n ...].
ADD	Accumulator is now $fib(n - 2)$, add $fib(n - 1)$ from the stack.
RET	The accumulator is now $fib(n)$ and the stack is clean. Return.

The outer loop computes the sum of the first N Fibonacci numbers.

LT-C 1; RET	Return if the accumulator i is less than 1.
PUSH	Push i onto the stack [i ...].
CALL-S "fib"; PUSH	Call $fib(i)$ and push result. Stack [$fib(i)$ i ...]
SWAP	Stack [$fib(i)$ i ...] becomes [i $fib(i)$...]
POP; ADD-C -1	Assign $i - 1$ to accumulator, [$fib(i)$...].
CALL-S "loop"	Call $loop(i - 1)$. Stack [$fib(i)$...].
ADD; RET	Return $loop(i - 1) + loop(i)$. Stack is clean.

The implementation is now a straightforward exercise. The virtual machine iterates over the instructions of algorithm and evaluates them according to the definitions of the instructions given earlier.

5.4.4 Performance

The complete program listing of the stack based virtual machine is shown on the next page: Listing 3. It computes the Fibonacci sum consistently in a little under 1.5 seconds. Interestingly, this almost identical to the time needed by the equivalent Python program. Recall that the previous implementation in C with a dictionary for function lookups needed a little under 1 second to complete.

Listing 3: Stack based Virtual Machine

```

#include <stdio.h>
2 #include <string>
#include <map>
4
enum instr { OP_LT_C, OP_RET, OP_POP, OP_ADD, OP_ADD_C,
6           OP_PUSH, OP_CALLS, OP_SWAP};
struct bytecode { instr i; /* instruction */ int op; /* operand */ };
8 int stack[1024]; int acc = 0; int* sp;
std::map<std::string, bytecode*> g_functions;
10
void interpret(bytecode* fn) {
12     for (bytecode* ip = fn; ip; ++ip) {
        switch (ip->i) {
14             case OP_LT_C:     if (acc >= ip->op) ++ip; break;
                case OP_RET:     return;
16             case OP_PUSH:     --sp; *sp = acc; break;
                case OP_ADD:     acc += *sp; ++sp; break;
18             case OP_ADD_C:    acc += ip->op; break;
                case OP_CALLS:    interpret(g_functions[std::string(ip->op)]); break;
20             case OP_SWAP: { int t = sp[0]; sp[0] = sp[1]; sp[1] = t; } break;
                case OP_POP:     acc = *sp; ++sp; break;
22             default:
                throw std::string("Unknown_bytecode");
24         }
    }
26 }

28 int main(int argc, char** argv) {
    sp = &(stack[1023]);
30     bytecode fib_bc [] = { {OP_LT_C, 2}, {OP_RET}, {OP_PUSH},
        {OP_ADD_C, -1}, {OP_CALLS, (int) "fib"}, {OP_PUSH}, {OP_SWAP},
32     {OP_POP}, {OP_ADD_C, -2}, {OP_CALLS, (int) "fib"},
        {OP_ADD}, {OP_RET}
34     };
    bytecode loop_bc [] = { {OP_LT_C, 1}, {OP_RET}, {OP_PUSH},
36     {OP_CALLS, (int) "fib"}, {OP_PUSH}, {OP_SWAP}, {OP_POP},
        {OP_ADD_C, -1}, {OP_CALLS, (int) "loop"}, {OP_ADD}, {OP_RET}
38     };
    g_functions[ "fib" ] = (bytecode*) fib_bc;
40     g_functions[ "loop" ] = (bytecode*) loop_bc;
    acc = 29;
42     interpret( loop_bc );
    printf("%d\n", acc);
44     return 0;
}

```

5.5 Naive Compiler

In order to improve on the results from the previous section the virtual machine is replaced with a simple JIT-compiler. At runtime the bytecode is translated into native machine code and after some bookkeeping that code will be executed.

5.6 Implementation

The compiler functions in three stages. First the bytecode is read, then the machine code is generated and finally the code is executed. The bytecode stage has not undergone substantial change compared to the virtual machine from the previous section. Executing the code after it has been generated is possible under two conditions: (1) the code has to be marked as executable in memory and (2) there must be a calling convention that defines how switch between C code and the code that has just been compiled. The first condition is dealt with in lines 53..55 of the program listing (see end of section). By allocating memory as `PROT_EXEC`⁷ the operating system is aware that the code is meant to be executable and no access violation will occur. The second condition is dealt with on line 12. It declares that the generated code will adhere to the *stdcall* calling convention which means that the callee is responsible for cleaning up the stack and that the value of all machine registers except for `eax`, `ecx` and `edx` must be preserved by the function.

Back of the envelope

When generating assembly code it is easy to lose sight of which factors are important. Considering once again the recursive Fibonacci algorithm, we observe that the case `return n if n < 2` can be described in 5 instructions and the second case in 10 instructions. By counting the number of times the function is invoked we determine that about 43 million instructions are executed in total during for the execution of program. The C version of the algorithm takes 18 milliseconds to complete on the same machine, which includes start up time and overhead from loading the standard libraries. From this we conclude that the time budget to evaluate a single high level instruction is 4.286×10^{-10} seconds, or $0.4286ns$

Processor speeds are often denoted in MIPS (millions of instructions per second) and the processor in the benchmark machine, a stock Athlon 2800+, has 7400 MIPS according to the Dhrystone benchmark. This is equivalent to $0.1351ns$ per instruction, so the back of the envelope estimate of $0.4286ns$ per instruction is pretty accurate.

To put this into context, accessing a CPU register takes a single cycle. Retrieving a value from the Level 1 cache takes a mere 2 or 3 cycles. If the Level 1 cache does not have the value the Level 2 is checked at a cost of 7 to 10 cycles. Retrieving a value from the RAM is an order of magnitude more costly: it can cost up to 200 cycles.

So the important thing to note from these numbers that it will not matter in the least if sub-optimal machine code is generated by the assembler. A cost of a few redundant or extraneous instructions is measured in nanoseconds, which is orders of magnitude below the cost of a few extra cache misses caused by bad locality.

⁷Assuming a POSIX system. Without a `PROT_EXEC` flag the kernel would refuse to execute the code as a security precaution.

The assembly code generated by the compiler has to be translated into 32 bit x86 machine code before it can be executed. The x86 architecture has a CISC⁸ design has hundreds of instructions, many of which take different forms. Since only a tiny number of very specific x86 instructions need to be compiled there is no need for a fully capable assembly backend. The different permutations of the needed instructions can be calculated manually using a reference manual. The "Intel64 and IA-32 Architectures Software Developer's Manual" series⁹ covers the instruction set in depth. The last part in the series "Optimization Reference" describes the relative cost of different instructions (including latency and throughput), interactions between the processor and the different cache layers and general best practices.

The machine code is generated using the x86 architecture reference guides. The complete program listing is included for completeness.

5.7 Performance

Program listing 4 contains the complete source code of the naive compiler discussed here. The source code is very similar to the virtual machine discussed previously (Listing 3) and as such is mostly self-explanatory.

The JIT-compiler prototype runs the same program in 26 milliseconds. Compared to the stack machine from the previous section, that needed 1.4 seconds, this is a 50x performance improvement. The implications of this result are discussed next.

⁸Complex Instruction Set Computing

⁹Available at <http://www.intel.com/products/processor/manuals/>

Listing 4: Naive Compiler

```

1  #include <stdio.h>
   #include <string>
3  #include <vector>
   #include <map>
5  #include <memory.h>
   #include <sys/mman.h>
7
9  enum instr { OP_EOF, OP_LT_C, OP_JMP, OP_RET, OP_INT_C, OP_PUSH, OP_POP,
   OP_ADD, OP_ADD_C, OP_CALL_S, OP_SWAP };
11 typedef int (*fnptr)(); typedef unsigned char modrm, uchar;
   std::map<std::string, fnptr> g_functions;
13 int __attribute__((stdcall)) print(int a) { printf("%d\n", a); return a; }
   struct bytecode { instr i; /* instruction */ int op; /* operand */ };

15 namespace x86 {
   enum reg32 { EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI };
17 enum preg32 { pEAX, pECX, pEDX, pEBX, pESP, pEBP, pESI, pEDI };
   struct emit
19 {
   struct jmp_block {
21     jmp_block(emit& e, int n) { offset_from = e.ocp - e.oc;
       idx_to = e.instr_offsets.size() + n; delta = (char*) e.ocp - 1; }
23     int offset_from; int idx_to; char* delta; };

25     std::vector<int> instr_offsets; std::vector<jmp_block> jumps;
       uchar *oc, *ocp; int inst_cnt;
27     void putw(int i) { memcpy(ocp, &i, sizeof(int)); ocp += 4; }
       void put(char c) { *ocp++ = c; }
29     void next() { instr_offsets.push_back(ocp - oc); }

31     void fix_jumptargets() {
       for (std::vector<jmp_block>::iterator it = jumps.begin()
33           ; it != jumps.end(); ++it) {
           *it->delta = instr_offsets[it->idx_to] - it->offset_from;
35     }
   }

37     void RET() { put(0xc3); };
39     void MOV(int i, reg32 r) { put(0xB8 + r); putw(i); }
       void PUSH(reg32 r) { put(0x50 + r); }
41     void POP(reg32 r) { put(0x58 + r); }
       void MOV(preg32 src, reg32 dest) { put(0x8B);
43         put(modRM(0, dest, reg32(src))); }
       void JMP_GE(int n) { put(0x7D); put(0xcc);
45         jumps.push_back(jmp_block(*this, n)); }
       void JMP(int n) { put(0xEB); put(0xcc);
47         jumps.push_back(jmp_block(*this, n)); }
       void CMP(int i) { put(0x3D); putw(i); }
49     void ADD(int i, reg32 r) { put(0x81); put(modRM(3, 0, r)); putw(i); }
       void ADD(reg32 r1, reg32 r2) { put(0x03); put(modRM(r1, r2)); }
51     void CALL(reg32 r) { put(0xFF); put(modRM(3, 2, r)); }

```

```

53     emit() {
        oc = ocp = (uchar*) mmap(0, 1024, PROT_EXEC|PROT_READ|PROT_WRITE,
55         MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    }

57     modrm modRM(char mod, char reg, reg32 rm) {
        return (mod<<6) + (reg<<3) + rm; }
59     modrm modRM(reg32 src, reg32 dst) { return (3<<6) + (src<<3) + dst; }
};};

61 void compile(const char* funname, bytecode* fn) {
63     x86::emit e;
    for (bytecode* ip = fn; ip->i; e.next(), ++ip) {
65         switch (ip->i) {
            case OP_JMP:    e.JMP(ip->op); break;
67             case OP_LT_C: e.CMP(ip->op); e.JMP_GE(1); break;
            case OP_PUSH:  e.PUSH(x86::EAX); break;
69             case OP_ADD:  e.POP(x86::EBX); e.ADD(x86::EAX, x86::EBX); break;
            case OP_ADD_C: e.ADD(ip->op, x86::EAX); break;
71             case OP_CALLS: e.MOV((int) &g_functions[ (char*)ip->op ], x86::EDX);
                e.MOV(x86::pEDX, x86::EBX); e.CALL(x86::EBX); break;
73             case OP_INT_C: e.MOV(ip->op, x86::EAX); break;
            case OP_SWAP:  e.POP(x86::EBX); e.POP(x86::ESI);
75                 e.PUSH(x86::EBX); e.PUSH(x86::ESI); break;
            case OP_POP:   e.POP(x86::EAX); break;
77             case OP_RET:  e.RET(); break;
            default: throw std::string("compile (...): unknown instruction");
79         }
    }
81     e.fix_jumptargets(); g_functions[ funname ] = (fnptr) e.oc;
}

83 int main(int argc, char** argv) {
85     g_functions["print"] = (fnptr) print;

87     bytecode fib [] = { {OP_LT_C, 2}, {OP_RET}, {OP_PUSH}, {OP_ADD_C, -1},
        {OP_CALLS, (int)"fib"}, {OP_PUSH}, {OP_SWAP}, {OP_POP},
89         {OP_ADD_C, -2}, {OP_CALLS, (int)"fib"}, {OP_ADD}, {OP_RET}, {OP_EOF}
    }; compile("fib", fib);

91     bytecode loop [] = { {OP_INT_C, 0}, {OP_PUSH}, {OP_PUSH},
93         {OP_CALLS, (int)"fib"}, {OP_SWAP}, {OP_ADD}, {OP_PUSH}, {OP_SWAP},
        {OP_POP}, {OP_ADD_C, 1}, {OP_PUSH}, {OP_LT_C, 30}, {OP_JMP, -10},
95         {OP_POP}, {OP_CALLS, (int)"print"}, {OP_RET}, {OP_EOF}
    }; compile("loop", loop);

97     (g_functions["loop"])();

99     return 0;
101 }

```

5.8 Conclusion

Table 2 summarizes the results of the performance of the different prototypes. The results are averaged over multiple trials and have insignificant variance. We reiterate that the performance numbers are merely indicative, and serve only to further guide the language design. However, the numbers support the assumptions made in the start of this chapter.

The almost identical performance of CPython and the virtual machine (of only 50 lines of code) suggests that a trivial model of a bytecode machine can accurately reflect what happens in a language environment as complex as Python. A simple model can be easily changed or extended for testing purposes, which all advantages that entails.

	Listing	Time (ms)	Performance Relative to GCC
GCC (Upper bound)	1	18ms	1
GCC with Dict	2	950ms	53
Bytecode VM Prototype	3	1420ms	87
Compiler Prototype	4	26ms	1
CPython 2.4	A.1	1570ms	87
Ruby 1.8 (Lower bound)	A.2	6050ms	336

Table 2: Performance Results

The second conclusion we draw from the data above is that even naively generated native code can lead to vast performance improvements: an intelligent optimizing compiler is not necessary.

The third conclusion we draw is that a compiler that emits machine code does not need to be significantly more complex than the equivalent bytecode machine. An extra compilation step is needed and the JIT compiler has to have awareness of different machine architectures for portability, and this inevitably leads to some extra complexity. However, given that modern dynamic languages comprise hundreds of thousands of lines of C code this extra complexity is negligible. It is encouraging that the compiler prototype, despite being only 100 lines of code long, outperforms Python by a factor of 60 and Ruby by a factor of 230, and is only 50% slower than GCC.

The fourth observation is that the performance of the Bytecode VM prototype and Python are almost identical. This means that other possible factors in VM performance, such as the use by Python of Arbitrary Precision Arithmetic, exception handling and garbage collection may not be that significant with regard to performance.

The fifth observation we make is that even a slight performance bottleneck has drastic implications. Colloquially speaking, a program is only as fast as its slowest part. Therefore we extend the JIT compiler from a starting point of being efficient (but lacking in functionality) towards a more capable JIT compiler while sacrificing a minimum in terms of performance. If an acceptable level of performance is maintained as new capabilities are introduced the end result will be, necessarily, one that is very efficient.

In the next chapter we describe two extensions to the prototype compiler: arbitrary precision integers and dynamic functions and methods.

6 Extending the Prototype Compiler

6.1 Introduction

The last chapter established that even a simple JIT-compiler can generate sufficiently fast machine code. In this chapter the prototype is extended in two parts.

The first extension replaces the simplistic integer model from the earlier prototype (that does not even check for numeric over- and underflow) with a more mathematically sound model.

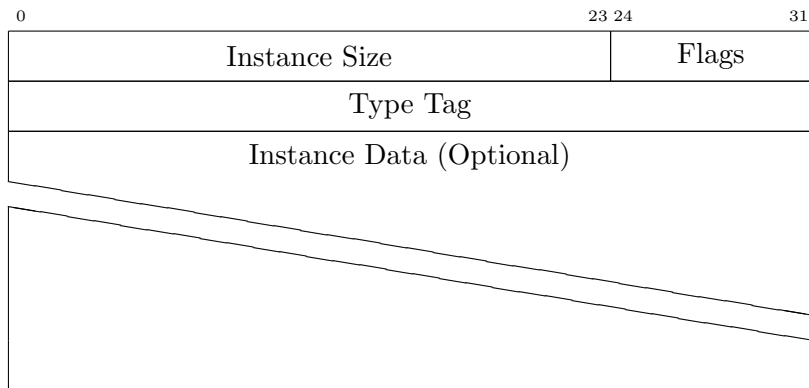
The second extension addresses the issue of efficient function and method invocations. In the previous chapter we demonstrated that repeatedly retrieving the address from a dictionary or hash-table has a significant impact on performance. Here we introduce the requirements and assumptions needed to maintain efficiency without sacrificing correctness.

Before we can proceed with these two extensions we quickly cover the basics of dynamic language design.

6.2 Language Design Preliminaries

Dynamic languages typically attach a type tag to every object instance. When a message is sent to an object the type tag of the object is inspected to determine which method to invoke. Since every object has to have a type tag the straightforward implementation sets this type tag whenever a new object is instantiated.

An object instance is represented in memory in the following manner:



- Instance Size** The size (in words) of the entire object. Used by the garbage collector.
- Flags** Various flags for internal bookkeeping.
- Type Tag** Points to the type of the object.
- Instance Data** Arbitrarily large area used for the object state.

Figure 2: Memory representation of object instance

A pointer to an object always points to the Type Tag. This substantially simplifies the logic in the virtual machine and JIT-compiler. Since the *Instance Size* and *Flags* have a fixed offset from the start of the *Type Tag* the values can be retrieved with simple pointer arithmetic by the JIT-compiler.

The *Instance Data* area is primarily used by native classes provided by the runtime environment. For instance, a String primitive stores the length of the string, its encoding, the size of the allocated memory and a pointer to the string itself in the *Instance Data* area.

The methods and member variables of user defined classes are all stored in a dictionary, a pointer to which is stored in the *Instance Data* area. User defined functions and methods cannot access the *Instance Data* area directly (or any of the other fields).

All objects are stored on the heap¹⁰. Dynamic languages have pass-by-value semantics and registers contain a reference to an object instance or a primitive value.

In order to improve the memory overhead and performance of primitive types and integer arithmetic in dynamic languages a number of common optimizations are made:

- Booleans are defined by a pointer to their type. Booleans are not instantiated like other values. The classes `FalseClass` and `TrueClass` define both the type of a boolean and their unique value.
- Small integers are encoded into the type pointer.
- Strings are interned, see definition 2.4 on page 3.

Integer Encoding

Allocating an object on the heap is wasteful when the integer is small enough to fit into a single register. By convention, a register in the runtime environment contains either a pointer to an object on the heap or it represents a small value. The difficulty lies in distinguishing between the two.

We observe that because heap objects are at least 4 bytes large every heap object is aligned on a 4 byte boundary. This implies that the last 2 bits of a pointer to any heap object must both be zero. These bits can therefore be used to store additional information.

The solution is then to encode small (30 bit signed) integers as odd numbers. Encoding an integer is straightforward: $N' = (N \ll 1) + 1^a$. This ensures encoded integers are always odd and the integer value can be retrieved with a single bitwise shift to the right. Every time a pointer is encountered the least significant bit has to be checked to determine whether the pointer refers to a heap object or whether it contains a small 31 bit integer.

^aThe \ll symbol denotes a logical bitwise shift.

¹⁰The memory area controlled by the garbage collector.

6.3 Extension: Efficient Integer Math

A typical virtual machine like those of Python and Ruby evaluate an integer addition in the following way:

Listing 5: Case analysis in typical Virtual Machine

```
1  case OP_ADD: // compute eax = eax + *esp
    ebx = *esp; esp++; // pop operand from stack
3    if type(eax) == INT && type(ebx) == INT {
        int64 tmp = decode(eax) + decode(ebx);
5        if (tmp > 2**30 || tmp < -2**30) { // overflow }
            eax = encode(tmp);
7        break;
    } else if type(eax) == BIGNUM && type(ebx) == BIGNUM {
9        bignum_add(eax, ebx);
        break;
11    } else if ( ... ) {
        // dictionary lookup for operator "+" and send message
13    }
```

The above logic can be classified into four distinct parts in the case where two integer operands are given:

1. type checking of the operands
2. encoding and decoding the operands
3. the overflow check
4. the numeric operation

In the JIT-compiler prototype from the previous section only part 4 was represented and the other 3 parts were neglected. In fact, the JIT-compiler prototype did not have any concept of type. Every value was assumed to be a 32 bit integer. Recall that in the Stack Based Virtual machine prototype the `OP_ADD` instruction had a trivial implementation. We observed that trivial virtual machine had comparable performance to CPython, which as shown by the pseudocode above, has substantially more complex logic for each instruction.

Jump Tables

A jump table takes advantage of the fact that opcodes are generally small sequential integers. An array is created that maps an opcode to the addresses of the corresponding case statement. With a jump table the jump addresses are calculated at runtime so they cannot be predicted easily by the CPU but it is still preferable to a simple switch statement where $N/2$ compares and jumps are needed on average to arrive at the correct case statement.

One problematic attribute of dynamic languages in the light of optimization is the absence of compile time information of the type of expressions. In the purest form of a dynamic language all objects are treated exactly in the same manner by the language environment and messages are sent back and forth between objects with absolutely no case analysis based on type. In practice virtual machines of dynamic languages pragmatically chose to treat some

primitive objects (such as integers and booleans) differently than more complex objects (for instance an object that represents a date and timezone). A virtual machine has specialized implementations for common operations, such as addition of two integers, of two floating point values and of arbitrary precision bignums. The virtual machine has to use several type comparisons every time an instruction is evaluated to determine if optimized code is available for the given operation. Each of these type comparisons is more expensive than the computation itself, which may be as trivial as adding two integers.

Most virtual machine implementations do not allocate small integers on the heap for performance reasons. Instead, small integers are encoded such that their type and the value both fit within a single register value. Typically, this optimization is based on the observation that memory is aligned by at 4 byte boundaries in memory. If all objects are known to start at multiples of 4 or then the two least significant bits of a pointer to an object must always be zero. These two bits can be used to encode meta-information about the pointer.

Boxing primitives

One of the downsides of the integer encoding scheme describe above is that the same technique cannot be applied to other primitive types such as *floats* or *chars*.

An alternative and more generic approach is to *box* and *unbox* variables. The idea is to store primitive values in a container, the box, that consists of a type and a value. When the primitive is boxed it can be treated like a high level object: it can be queried, it can respond to messages and so on. When the variable is used locally the box is discarded and the value is again a primitive, with all the performance advantages that come with it. In languages with a static type system, such as C#, boxing is straightforward to implement as the compiler can easily determine that a primitive value is only treated as such. This is more difficult for a dynamic languages where no such guarantees exist.

The third class of overhead for simple integer additions is the overflow check. Ordinarily no hardware exceptions are thrown on overflow, so after every integer operation the virtual machine has to explicitly check for over- and underflow. When an overflow is detected an exception can be thrown or the numeric object can be promoted to an object that has sufficient bits to store the result. These assertions are wasteful because integer overflows happen only very rarely but the check are made after every single operation even when both operands are tiny and no overflow could possibly occur.

We now investigate under which conditions these three classes of checks can be eliminated during JIT compilation.

6.4 Eliminating type checking

A single type assertion is not an expensive operation. However, when a virtual machine has to do a dozen type comparisons to determine how to evaluate a single instruction this can become a significant issue for the performance of the virtual machine. A JIT-compiler cannot do any better in the complete absence of type information: the exact same case analysis from the virtual machine is needed, so the computational overhead remains.

One of the benefits of JIT-compilation is the elimination of the overhead of *opcode dispatch*. The native code generated by the JIT-compiler can be executed linearly from top to bottom. Every fat instructions can be split up into a sequence of *lean instructions* without

Assembly Bloat

During the compilation process every bytecode instruction is substituted with the equivalent assembly code. If a bytecodes is fat a lot of assembly code needs to be generated in its place. A single bytecode may be compiled into a hundred bytes of assembly. This explosion in program size is referred to as *assembly bloat*. It degrades performance and makes the generated assembly code difficult to read. A *bloated* program contains many code paths that will never be taken (because even dynamic programs are *type-stable*).

The bloating problem can be solved by replacing all occurrences of a fat instruction with a function call to a single implementation of that instruction. This results at a compiled program where the CPU has to make a function call for every instruction, just as if it were a virtual machine. A function call is still preferable to a jump-table based dispatch strategy because the address of each function call is fixed into the generated assembly so *branch target mispredictions*^a cannot occur.

^aModern processors look at branch targets ahead of time to improve the flow of the instruction pipeline.

loss of efficiency. Suppose there is a fat `OP_ADD` instruction that internally distinguishes between addition of integers, floating point values and strings can be split up into a sequence of *lean* instructions such as type comparisons and branches that invoke type-specific addition instructions. If the compiler knows the types of the arguments supplied to the `OP_ADD` instruction all of the lean instructions can be optimized away except for the lean instruction that processes the actual addition.

The type check itself does not become expensive when integers are encoded. If a pointer is odd it is an encoded integer (by convention) so the type has been determined. On the other hand, if the pointer is even it must refer (by convention) to an object. Every object must contain a pointer to the class it is an instance of, so the language environment can retrieve that slot to determine the type.

6.4.1 Type assertion elimination via type inference

Previous attempts to optimize dynamic languages via type inference have been mostly ineffective, as mentioned in chapter 2. Type inference in a dynamic language is difficult when the majority of expressions do not have a trivially determinable type. A type inference engine may, naturally, classify every expression as having the type `+Object`¹¹.

Type inference is also difficult for dynamic languages because type unions are needed (to indicate an expression has either type A or type B). If an expression has two subexpressions both of which are known to have one of two types then the type inference engine has to enumerate all possibilities: the Cartesian product of the types of the subexpressions. The type inference engine can always fall back on the generic `+Object` type classification, but this will make all attempts of further type inference in that area fruitless: meaningful optimizations can only take place when narrow types can be derived.

A third problem with type inference is that the types, even if they can be accurately determined, are very complex in dynamic language. As shown by "A core Calculus for Scala

¹¹Where `+` denotes covariance, meaning that the expression either has exactly the type `Object` or a more specific type thereof.

Type Checking” a type inference engine has to distinguish between invariant, covariant and contravariant types (which is impossible without manually inserted type hints) in order to correctly guide the JIT-compiler.

For example, the simple expression `[Person("Bob", 46), Person("Jane", 31)]`, can have multiple types:

- `Array<+Person>`: An array of objects of type `Person` (or subclasses thereof). Adding a `Child("Zed", 4)` to the array is allowed.
- `Array<Person>`: An array (but not a subclass of `Array`) of instances of `Person`. Adding a `Child` object is now prohibited.

Essentially, a type inference engine cannot determine intent of the programmer. Type hints can be used to clarify these ambiguities, but this partially defeats the purpose of using a dynamic language. We conclude that classical type inference is not very suited for dynamic languages. In the Related Research section a few approaches to dynamic language optimization based on type inference are covered.

6.4.2 Type assertion elimination via execution traces

A tracing JIT compiler can remove redundant type assertions by inspecting the types of the variables while the program is running, either globally or for sections of code that take the computation time. The assumption when recording a trace is that the next time the same piece of code is executed the variables and expressions will be of the same type as before. This means that for every expression there now is a type assumption. The function (or code segment) in question is JIT-compiled again, and type assertions are inserted for the function parameters, global variables that are referenced, after method calls, and so forth. If one of those type assertion fails at runtime the less efficient JIT code is used from that point on. As long as the type assertions hold the type conditionals for every numeric opcode can now be safely removed. The program is assumed to be in *SSA* form: variables are only assigned once. This implies that at most a single type assertion is needed per variable.

To illustrate which kind of type assertions can be eliminated with this approach we will use this trivial example program that consists of a for-loop a few additions and a function call:

```
1  n = 0
   for i in 0..N do
3     f = factorial(i)
       n += i + f
5  end
```

The reader can intuit that the variables are all integers and that a type assertion is only really needed to determine the return value of the *factorial* function. When the JIT-compiler evaluates the first loop body for the first time it determines that the variables are indeed all integers. Type assertions are inserted after every assignment. Since there are three variables and each is assigned once this means only three type assertions are placed into the code. This is a substantial improvement over the eight type checks used before. Since a variable is referenced at least once after assignment (otherwise the assignment can be eliminated entirely) it is always more efficient to determine the type just once after assignment instead of checking the type for every operation on the variable.

This technique also has a number of downsides. First, it may be difficult to determine which code fragments are suitable candidates for trace based optimization. Second, the instrumentation and optimization process should happen in parallel to the execution of the program to keep from interfering with the normal program execution. Third, it adds a substantial amount of complexity to the compiler.

6.4.3 Type assertion elimination via static analysis

When a software engineer looks at a code fragment written in a dynamic language he can generally make an educated guess about the types of most of the variables and (sub)expressions. This observation is the basis of the static analysis approach to type assertion elimination. During the JIT-compilation process every function is compiled twice. Once completely pessimistically (with all type checks in place) and once with optimizations in place based on the *assumed* types of the variables.

Note that correctness is not a must for the static analysis: if an assumption turns out to be wrong the execution will continue with the less efficiently (but assuredly correct) version.

Consider these simple rules for static analysis:

1. after assignment the rhs and lhs have the same type
2. the type of a variable doesn't change after it has been assigned
3. both arguments of numeric (infix) operations have the same type
4. functions always return objects of the same type

These rules are sufficient to sensibly determine the types of the variables of the program from the previous section.

1	<code>n = 0</code>	<code># type(n) = int, rule(1)</code>
	<code>for i in 0..N do</code>	<code># type(i) = int, rule(1)</code>
3	<code> f = factorial(i)</code>	<code># type(f) = ?</code>
	<code> n += i + f</code>	<code># type(f) = int, rule(1, 2, 3, 4)</code>
5	<code>end</code>	

Because the function is JIT-compiled when it is called for the first time a lot of information about the environment is readily available. The types of the function's arguments can be inspected, the types of global variables can also be retrieved. The static analysis engine can even peek inside objects used in the body of the function to make an educated guess about types.

This approach has a number of advantages over the trace based approach. First, it's deterministic and the programmer can predict which assumptions the static analysis engine will make. Second, the programmer can insert type assertions at tactical places to guide the static analysis engine. Third, if a type assumption proves to be wrong the analysis engine can take that into account and try again at a later time. Fourth, a naive version is straightforward to implement. Fifth, this technique can be applied for every operation where type checks are used. For instance: when invoking a message on an object (e.g. `a.calculate(b)`) ordinarily a dictionary lookup is needed to discover the address of the `calculate` method, every time the statement is evaluated. If the type of variable `a` can be guessed by the JIT-compiler it can then lookup the address of the `calculate` method and insert it into the code together

with a type assertion. On evaluation, if the type matches what the compiler guessed then the method can be invoked directly (without a dictionary lookup) and otherwise the runtime environment can still do a dictionary lookup to discover the correct method to call.

6.5 Eliminating integer encoding and decoding

By adding the the integer encoding logic as described on Page 31 to the bytecode virtual machine the impact of integer encoding on performance can be measured. The results are in the following table:

	Time (ms)
Bytecode VM Prototype	1420ms
Bytecode VM Prototype with integer encoding 1	1460ms
Bytecode VM Prototype with integer encoding 2	1220ms

Table 3: Performance Results

The first row repeats the results from the original bytecode virtual machine. The second row represents the same virtual machine but with integer encoding logic added for every integer instruction. An addition now consists of decoding the two integer arguments, performing the addition and re-encoding the result. It turns out that this additional logic has no meaningful impact on the performance of the virtual machine as a whole. Interestingly, when taking the integer encoding logic one step further, by performing operations on encoded integers without first decoding them (see sidebar) the whole process becomes faster than the original benchmark, even though the virtual machines are virtually identical.

From these results we conclude that integer encoding and decoding steps have no meaningful impact on performance. The JIT-compiler can therefore adopt the same encoding and decoding strategies as the prototype virtual machine did and as most virtual machines for established dynamic languages do.

6.6 Eliminating overflow checks

Eliminating overflow checks is difficult, because the overflow assertion can only be eliminated when the JIT-compiler can prove the check is redundant. Since it depends on the value of variables which are produced by arbitrary expressions the problem of removing overflow checks with perfect accuracy is undecidable.

Instead of removing the overflow assertions altogether the JIT-compiler can place a single overflow after all end points of a loop. If an overflow occurs the program state will be rolled back to the point of last successful overflow check. The overflow assertions are placed back in the machine code of the loop and the program resumes execution. This time when the overflow happens the integer is promoted (or an exception is thrown). Although this may work in theory, there are a number of practical concerns. First, rolling back the state can be (very) difficult. Second, if a lengthy operation is rolled back the performance may degrade instead of improve. Third, IO operations and system calls cannot be rolled back. For these reasons this approach has not been investigated further.

A better approach to reduce the number of redundant overflow checks is by means of range analysis. When a function is JIT compiled the ranges are computed of the variables for which no overflow check is necessary. The range analysis can happen locally, on the function level.

Manipulating encoded integers

Since the integer encoding is such a simple mathematical transformation ($N' = 2N + 1$) it allows for manipulation of the encoded integers without decoding them beforehand. Suppose we wish to add a regular integer m to an encoded integer N :

$$N' = \text{encode}(\text{decode}(N) + m) \tag{1}$$

$$= \text{encode}((N \gg 1) + m) \tag{2}$$

$$= (((N \gg 1) + m) \ll 1) + 1 \tag{3}$$

$$= ((N \gg 1) \ll 1) + 1 + (m \ll 1) \tag{4}$$

$$\{N \% 2 = 1\} \tag{5}$$

$$= (m \ll 1) + N \tag{6}$$

If m is a constant known at JIT-time the bitshift can be precomputed and the addition instruction reduces to a single `add %eax m` operation. If the integer m happens to be encoded the bitshift is no longer necessary: $N' = (M - 1) + N$. For integer multiplication of an encoded integer N and an ordinary integer m the solution follows similarly: $N' = \text{encode}(\text{decode}(N) * m) = (N - 1) * m + 1^a$. Although only a few x86 instructions are saved with this technique it is roughly twice as fast as explicit encoding and decoding. Given how prevalent integer operations are in any program this optimization is probably worth doing.

^aThe alternative $N' = \text{encode}(\text{decode}(N) * m) \stackrel{?}{=} N * m - (m - 1)$ does not hold, since the intermediate value $N * m$ may yield an overflow that would otherwise not occur.

The functions can be JIT-compiled multiple times so the JIT compiler can take advantage of the most specific runtime information. Sol et al[20] successfully eliminate a large number of overflow tests by static means and via trace analysis, but the increased JIT-compilation time leads to a net negative performance result. The authors note that their algorithm, being a form of range analysis, may be categorically too slow to be suitable for a JIT compiler.

Purely static approaches for elimination of overflow elimination may be more suitable. Su and Wagner[21] have presented an optimal and polynomial time algorithm that finds optimal range constraints on variables based on a Static approaches like these depend to a large extent on static type information on variables and expressions, which is mostly unavailable.

The conclusion is clear: elimination of overflow tests is computationally expensive and there are only little performance gains to be made when successful. Consequently, overflow tests need be inserted after every integer operation. One small optimization can be made, however. By defining the integer encoding, integer decoding and the overflow test as separate instructions the JIT compiler can eliminate redundant instructions in specific situations that are known to be safe, such as with operations on the iterator variable in an ordinary for loop.

6.7 Conclusion Integer Math

The goal was to implement numeric operations with a minimum of overhead, without increasing the complexity of the compiler significantly and without sacrificing much in terms of language dynamicity. We observed that in a dynamic language few¹² statically typable

¹²Literal expressions are statically typed even in a dynamic language.

expressions the compiler is unable to make observations based upon which it may optimize numerical operations into efficient native code. Arbitrary precision numerics added a second layer of difficulty.

The internals of several virtual machines were inspected and the functionality was compared to the functionality of the trivial virtual machine from Chapter 5. Several potential sources of overhead were identified, such as repeatedly encoding and decoding integer, checking for integer overflow and type checking.

We have shown that encoding and decoding integers bears virtually no cost. The benefit is that encoded integers do not have to be allocated as objects on the heap. It may seem that decoding two integers, doing a mathematical operation and finally encoding the result is substantially slower than doing only the mathematical operation itself but our measurements show that this is not the case. This result has been attributed to the ability of modern processors to evaluate multiple micro-operations in parallel.

Overflow tests turn out to be relatively inexpensive and the elimination of these tests via range detection is effective, but computationally too expensive[20]. A JIT compiler may only allocate few resources for the elimination of overflow tests even though the algorithms to detect unnecessary overflow tests are complex and computationally expensive.

7 Extension: Efficient functions and methods

In this section we cover the different implementation strategies for efficient function calls and method invocations. The difference between a *function call* and *method invocation* is that a function is identified uniquely by its name (and perhaps a namespace), whereas a method is identified by a name and the type of the object on which the method is called. During this section it is assumed that both methods and functions may be defined and re-defined at runtime. Given that more elaborate function calling strategies such as multiple dispatch can be implemented with ordinary functions or methods, so they will receive no further attention.

The runtime environment of a dynamic language typically handles a function call via a lookup in a global hash-map or dictionary every time the function is called. A method is dispatched in a similar way, every object has a dictionary that maps a message (a string) to a function object. When a message is sent to an object the method dictionary of the object is inspected. If a function is found it can be called, otherwise the language environment checks if the parent classes of the object respond to the message. If no handler can be found an exception is thrown or a "no-method" handler is called instead (see box), so the object can resolve the situation locally.

Aspect Oriented Programming

A no-method handler method is useful for techniques such as aspect oriented programming. Suppose a message `benchmark_docalc` is sent to an object, but no corresponding method exists. The no-method handler can then check if the message starts with `benchmark_` and if so, start a timer, invoke the `docalc` method, stop the timer and output the time spent in the `docalc` method.

The optimizations described in this section exist in addition to the method dictionary. This is because in a dynamic language any object can query which messages it responds to, which slots it has, and so forth; the dictionary cannot be optimized away. To further illustrate this consider the example program below. A clever compiler can theoretically determine ahead of time where the `now`, `methods`, `select` and `each` methods reside and optimize these function calls. No such optimizations are viable on line 6, where a message `m` is sent to the time object, but no assumption can be made with regards to the value of `m`.

Typical usage of introspection in a dynamic language

Listing 6: Introspection in Dynamic Languages

```
1  t = Time.now # get current time
   # select the methods that start with 'h'
3  t.methods.select{|m| m[0..0] == 'h'}.each do |m|
   # invoke the method (0 args) and print
5  puts "method_%.s() =>%.s" % [m, t.send(m)]
   end
7  # output:
   # method hour() => 16
9  # method hash() => -857433557
```

We cover the optimization of function calls and method invocations separately and compare the different optimization strategies at the end of this chapter.

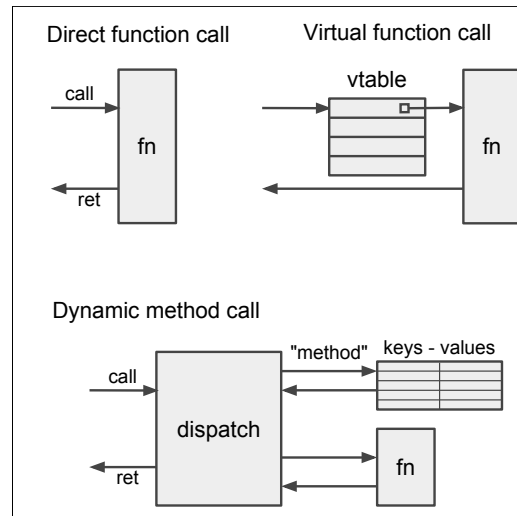


Figure 3: Types of function calls

7.1 Optimizing Function Calls

Three different approaches are covered here. Two approaches that take advantage of the fact that functions are redefined only occasionally and that therefore the function address can be assumed to be stable. The third approach focuses on optimizing the dictionary lookup itself.

7.1.1 Solution 1: direct function call

The idea is conceptually simple. When a function call is being compiled the name of the function is known because it shows up in the *AST*. Consequently, the compiler can discover the address of the function based on the name and insert it directly into the assembly code.

Although this works in the majority of cases, two problematic areas remain. First, the function may not yet exist at the time of compilation, which means there is no address to insert into the code. The simplest solution is for the JIT-compiler to leave out the optimization in this case.

The second problem is that when a global function is recompiled it will invariably be relocated in memory as well. This means all code fragments that refer to the function by address will have to be updated. To update these addresses efficiently a "called by" list is maintained for every global function. This called-by list will contain all the locations in memory from where the function in question is called. When the function is redefined and its address changes the called-by list can be used to update all addresses to the new location.

Advantages and downsides

Doing the dictionary lookup ahead-of-time has a number of advantages and downsides. The first and foremost advantage is that the generated code is optimal with regard to performance: a direct jump is optimal. It's also a conceptually simple optimization that can be implemented in a straightforward fashion. The only overhead resides in the called-by list, and the overhead is a fraction of all the other data structures that are in use during compilation. In the worst case scenario a few hundred jump addresses have to be fixed when a function is redefined.

The major downside to this approach is that the called-by list may never contain obsolete references. This implies that before a function is redefined all entries in the called-by lists must be removed. This requires the introduction of yet another data structure per function: a callee list. So per function the compiler must keep track of which functions it calls and by which functions it is called. In other words: more bookkeeping and more complexity. Arguably the compiler ought to keep track of this information regardless: caller and callee lists make for valuable debugging aids.

7.1.2 Solution 2: indirect function call

The complications of the first solution arise from embedding the address of the function into the object code. Here we explore the logical next step: introduce a level of indirection so a function can be safely recompiled or moved in memory. The JIT-compiler will place a pointer to a pointer to a function in the object code so the location of the function itself is no longer important.

Advantages and downsides

The trade-off is clear. The runtime environment no longer has to patch code when a function is moved or redefined and in exchange a pointer has to be dereferenced before every function call.

A beneficial side effect of the indirection simplifies the compilation of mutually recursive functions. If a function has to be compiled before the function's address is known a depth-first compilation strategy is necessary. For mutually recursive functions this cannot work, for neither function can be compiled before the other. Consequently, the JIT-compiler will have to use placeholder addresses and patch those addresses after a necessary functions are compiled and just before execution of the program continues. When using a level of indirection for function call this problem disappears; functions can now be compiled in any order the JIT-compiler finds suitable.

A downside of this approach is that if every occurrence of a function call calls the same indirect function no optimizations or specializations (based on e.g. argument types) are possible. So the essential problem remains: the generated object code has to be patched when the JIT-compiler decides a different specialization of a function is more suitable.

7.1.3 Solution 3: *vtable* based virtual function calls

Some programming languages, such as C++, implement dynamic dispatch by means of a *vtable*. A *vtable* is an array of pointers to member functions in the class hierarchy. The combination of a *vtable* (known at compile-time and derivable from the type of the object) and a fixed offset are sufficient to dispatch the virtual function at runtime. Classes that belong to the same class hierarchy have the same *vtable* layout, i.e. a polymorphic method has the same offset in every *vtable* in the class hierarchy.

Since the *vtable* approach is efficient and straightforward to implement, we investigate if this same approach can also be used by a dynamic language. For virtual dispatch in C++ to work the classes need to have a shared base class and the type of either the child or base class has to be known at compile-time. In a dynamic language every object usually has a shared base class (usually called *Object*). The solution then is to have a single *vtable* for all objects and all methods and to use that to dispatch the polymorphic functions calls.

Advantages and downsides

Unfortunately this approach is not viable. A *vtable* has to be contiguous in memory and the *vtable* layout has to be shared by every class. This means that whenever a new function is defined every parent and child class has to be updated to reflect that the class does not override the newly defined function. Whenever space runs out in a *vtable* for a new entry the consequences are even worse, this time every *vtable* has to be reconstructed and every function call has to be patched with the location of the new *vtable*.

From this we conclude that a *vtable* based technique is simply not viable.

7.1.4 Solution 4: self-patching direct function call

A combination of the approaches outlined above is also possible: a self-patching direct function call. Functions are called directly; the JIT-compiler inserts the address of the function directly into generated code. When the function is redefined a small fragment of code is stored at the old memory location. This fragment of code, when executed, will patch the code the function that called it so that it refers to the new definition of the function. As a result, all references to the old version of the code will be patched lazily.

Advantages and downsides

The main advantage is that it combines optimal performance in the most common scenario (regular function call) with the flexibility of redefining functions at will at runtime. The implementation is straightforward: logic for walking the call stack is already needed to collect stack traces, so this does not lead to any additional complexity. The logic for replacing the address of the old function with the new one is also straightforward.

Conclusion

The fourth and last solution offered is significantly superior to the others for our purposes. Function calls are optimal in performance and functions may be redefined freely. The alternative approaches are either inefficient, too complex, or both.

7.2 Optimizing Method Invocations

The techniques described above for global functions do not work for method invocations. Method invocations can be polymorphic, which means that the method that is invoked depends on both the type of the object and the message itself. If the type of the object is not known during JIT-compilation a dictionary lookup to retrieve the method by name is necessary. Here too we observe that in the vast majority of the time a method literal on a specific line of code will refer to the same method every time the code is executed[10]. Functionality is implemented in terms of class hierarchies and methods, even when no polymorphism is needed or even desired. Classes and class hierarchies are often used as namespaces and to group functionality into cohesive units, not because methods need to be dispatched based on the type of the instance.

Although the process of determining which method to call is referred to as a dictionary lookup, the real algorithm is slightly more involved:

Recursive algorithm for finding a method

Listing 7: Recursive Method Search Algorithm

```
1  fnptr* find_method(Obj* obj, string msg) {
2      if (!obj) return raise_exception("bad_operand_obj");
3      if (obj->method_dict.has_key(msg)) {
4          return obj->method_dict[ msg ];
5      } else if (obj->parent_class != obj->class) {
6          return find_method(obj->parent_class, msg);
7      } else {
8          return raise_exception("method_not_found");
9      }
10 }
```

In the worst case scenario all parent classes of an object have to be traversed before the correct method is located. It is possible to expand the method dictionary of a class with all the methods defined by its ancestors, such that a single dictionary lookup is sufficient to ascertain which name corresponds to which method. The obvious downside to expanding method dictionaries is that adding or redefining a method of a class high in the class hierarchy (e.g. the ubiquitous *Object* class) requires that the method dictionary of child classes are updated to reflect the change. There may be easily be thousands of classes in a program, so updating mutating a class can become a very expensive operation.

We describe several approaches for improving on the naive algorithm above the rest of this section.

7.2.1 Caching

The observation made earlier was that if a statement is evaluated multiple times in all likelihood the same methods will be invoked every time. This suggests that caching which method was invoked may speed up future method calls, provided that the runtime environment can efficiently check whether a given method is the correct one.

Inline Cache of the Method Address

Listing 8: Inline caching of method address

```

2  if (type(a) == first_type && *cached_fn) {
    (*cached_fn)(args);
  } else {
4  fnptr fn = find_method(a, "foo");
   fn(args);
6  if (cached_fn != fn) {
    first_type = type(a); # patching the code
8  cached_fn = fn;       # patching the code
   }
10 }

```

The JIT-compiler will output assembly code with similar logic as shown in the pseudo code above for every method call. Whereas an ordinary function is identified only by its address, a method is identified by its address and by a type. So if a message is sent to two instances of the same class then then the same method has to be invoked. The message is generally known at compile time, as it appears as a literal in the code. The unknown is the type of the instance, which will simply be cached, along with the address of the method.

The difficulty of caching method addresses lies with cache invalidation. The simplest solution would be to simply invalidate the entire cache every time a new method is defined. According to Zakirov et al[25] this can become very detrimental to performance: the cache can be very large for complex programs. More granular cache invalidation is therefore desirable.

The different situations that have to be covered are:

1. a new method m is defined for class C ,
2. a method m is redefined,
3. a method m is defined that is already defined for a base class P ,
4. a method m is removed.

The method cache can be defined as `dict<string, dict<class, cell>>`. A method name maps to a dictionary of classes to cells. The cell can be a method or a class variable or any other property that belongs to the class. Defining a new method and redefining a method is semantically the same because absent methods can be considered methods that are mapped to the no-method handler. When a method is redefined the cache for m can be updated (or invalidated) for class C and `descendants(C)`. When a method is already defined for a base class the cache for m also has to be invalidated for `ancestors(C) - ancestors(P)`. Removing a method is equivalent to assigning it the no-method handler, so this case is covered by (2).

The class hierarchy is flat in practice (there may be tens of thousands of classes, but a single class will not have more than a dozen ancestors), so invalidating the ancestors cells is inexpensive. Calculating the set of *descendants* is easy (provided every class has a list of its direct descendants), but the set may be large. In fact, every class is expected to be a descendant of the root class *Object*. So iterating over every descendant class and invalidating the corresponding cell (if it exists) is not a viable approach. An alternative is to iterate through the dictionary that corresponds to the method name, and to check whether the class is in the set of descendant classes. This is not always viable either, because constructing the set of descendant classes is an abundant amount of work.

There is a reasonable compromise, using a simple heuristic. If there are few items in the dictionary that maps classes to cells, just invalidate it entirely. Otherwise, if a class has too many¹³ descendants then stop computing the set of descendants and invalidate the entire dictionary for that message name. If a class has few descendants, then do the "optimal" algorithm where only the correct cells from the dictionary are invalidated.

Driesen[10] covers in "Software and Hardware Techniques for Efficient Polymorphic Calls" a number of general algorithms to improve the performance of polymorphic method calls in both static and dynamic languages. He notes that the hit ratio of an inline cache as described above has a hit-ratio between 90-99% for a typical Smalltalk program. An inline cache is vulnerable to worst case behavior with a 0% hit ratio. This can be remedied by replacing the Inline Cache with a Polymorphic Inline Cache. A polymorphic inline cache has multiple slots in which to store the combination of class and corresponding method. When a cache miss occurs a complete lookup is performed and the result is appended to the inline cache. According to [10] resolving cache misses in a inline cache can take up to 25% of a program's processing time and a polymorphic cache eliminates the cache miss problem in practice.

7.3 Conclusion Functions and Methods

The runtime cost of determining which method is called under which circumstances cannot be eliminated entirely, but caching the results reduces the complexity in the most common scenario to a simple type check. The most appropriate cache consists of a (Polymorphic) inline cache per method invocation and a global method cache in the event an inline cache miss occurs. The importance of granular cache invalidation has been raised by[25] but the impact of specific invalidation strategies cannot be measured with a small prototype language, as the negative effects are meant show up in large complex and highly dynamic projects.

¹³Depends on implementation details; but 10 is may a reasonable number.

8 Design Final Prototype Language and Compiler

In the previous chapter two major subjects have been addressed: the consequences on compiler design with regard to efficient integer math and the consequences on compiler design with regard to efficient functions and methods. After establishing the basic framework and context in which the two subjects were addressed and after observing the extent in which these two subjects affect the compiler design of a dynamic language, a number of conclusions were drawn.

For the problem of implementing efficient integer math for the JIT compiler we concluded that encoding small integers has the big performance benefit compared to the alternative approaches. We concluded that overflow tests are a small but necessary cost for correctness and that overflow tests cannot be profitably removed.

We emphasized that the level in which the JIT compiler can make performance optimizations is greatly constrained by computation time (the cost of the optimization must not exceed the proceeds) and by the lack of reliable type information. Several approaches to determine types or to eliminate type checks have been touched upon: type test elimination via type inference, via execution traces and via static analysis.

An optimal solution in terms of runtime performance for dispatching functions was presented, based on an inline cache of the function address and by replacing a function with an adaptive code fragment that lazily fixes bad cache values upon redefinition of a function.

For efficiently dispatching polymorphic methods a number of techniques have been considered. An Polymorphic Inline Cache (PIC) in combination with a global lookup cache gives a straightforward and reasonably efficient solution for the lookup of polymorphic methods. This is the strategy used by Smalltalk and Self compilers [23].

This chapter will focus on the remaining design decisions for the development of the JIT-compiler. A suitable garbage collection algorithm will be picked, a better system for code generation will be described and the language semantics will be defined.

8.1 Design Decisions

There are many factors that affect the direction of language design. The garbage collection algorithm affects how much memory overhead allocation has and whether objects may move in memory. This in turn affects whether distinguishing between stack and heap allocation is worthwhile. If stack-allocated variables are introduced this may affect the language grammar or it may determine whether objects are assigned by-name or by-value. Essentially, nearly every aspect of a programming language has major influence on every other attribute of the language. So when discussing the various attributes and decision decisions in the rest of this chapter the dependency between the parts of the programming language are not exhaustively enumerated. All different combinations have been carefully considered, however.

The following design decisions are covered in depth in the rest of this chapter:

1. Garbage Collection
2. Code Generation
3. Variable Lifetime

8.2 Garbage collection

The prototype language does not make any specific demands on the method of garbage collection. The garbage collector may move objects in memory if it so desires (*moving collection*). Complete runtime information is available, so the garbage collector may use an *exact* algorithm, where objects are deallocated if and only if they are unused.

The information in this section is primarily based on *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* by R. Jones and R. Lins[16] a comprehensive collection of garbage collection algorithms.

The following algorithms are deemed most suitable for use in a simple just-in-time compiled dynamic language:

- Mark-Sweep Garbage Collection,
- Mark-Compact Garbage Collection,
- Copying Garbage Collection,
- and Generational Copying Garbage Collection,

In addition we will shortly consider *Reference Counting*. Although reference counting is strictly speaking not classified as a garbage collection algorithm it can be used instead of garbage collection and as such is worth consideration.

8.2.1 Reference Counting

A reference counting strategy uses a counter that tracks via how many locations an object can be reached. If this counter reaches zero the object is considered unreachable and can therefore be deallocated immediately and the memory can be returned to the memory allocator.

The advantages of reference counting are that the process is deterministic, that an object is deallocated the moment it becomes unreachable and the cost of garbage collection is distributed evenly over the runtime of the program: there are no expensive garbage collection phases. The disadvantages are that extra memory has to be allocated for every object for the reference counter itself, circular data structures cannot be deallocated easily, and significant memory fragmentation may occur.

The CPython interpreter uses a reference counting as its primary means of garbage collection.

8.2.2 Mark-Sweep Garbage Collection

When allocation fails because of an out-of-memory condition the garbage collection algorithm is invoked. The Mark-Sweep Garbage Collection approach works in two stages. First the entire memory area is scanned and all accessible objects are marked as such.

The mark stage works in the following manner: the language environment assumes that all global variables, all variables referred to by registers and all local variables are accessible. The objects referred to by these variables are known as the *root set* and are marked. All objects accessible from this root set are traversed and also marked as in-use, by traversing the forest of objects in a depth-first or breadth-first manner.

During the sweep stage all objects that are not marked are deallocated and the memory allocator may now reuse the memory. The mark is removed from all objects that are marked in preparation of the next mark-sweep cycle.

The advantages of mark-sweep garbage collection are that the implementation is trivial, there is no per-object memory cost and that garbage marking time is linear in the number of *accessible* objects. The disadvantages are memory fragmentation, and poor locality.

8.2.3 Mark-Compact Garbage Collection

A mark-compact garbage collector has the same mark phase as a mark-sweep collector. In the second phase all accessible (reachable) objects are moved so that they are consecutive in memory. For this to work all pointers to moved objects have to be updated. This can be accomplished by using a table to keep track of the old and new locations of every object and updating pointers where necessary. Because this compacting phase is more computationally expensive than a regular sweep phase compaction only happens occasionally, for instance when the process is running out of memory or when fragmentation has exceeded some limit.

This garbage collection algorithm is substantially more complex than the regular Mark-Sweep algorithm but the downside of fragmentation is solved at substantial run-time cost. Locality may still be poor, however.

8.2.4 Copying Garbage Collection

A copying garbage collector divides the working memory into two equally large areas. The memory allocator will allocate memory sequentially from the first area. When the memory of the first area has been exhausted the copying phase starts. Like with the marking phase in the previous algorithms, the *root set* of objects and objects reachable from the root set are traversed and immediately copied sequentially onto the second area. This automatically compacts the objects. As with the Mark-Compact algorithm, all existing pointers have to be updated to refer to the objects' new locations. After copying phase has finished new objects are allocated from the second memory area and the first memory area is now considered empty.

This is a substantial improvement over the naive mark-compact algorithm but at a significant cost: memory usage is doubled. Additionally, this algorithm will copy all memory in use by the program whenever an out-of-memory condition occurs. A copying garbage collector does not suffer from the fragmentation and memory locality issues.

8.2.5 Generational Copying Garbage Collection

As a performance optimization additional memory pools may be introduced. These memory pools are called *generations*. When new object is allocated it is allocated from the first memory pool (youngest generation). When this first memory pool reaches an out-of-memory condition all objects are promoted and copied to the second generation memory pool. When this second memory pool has insufficient space to store the reachable objects of the first memory pool a second copying phase is initiated where the contents of the second memory pool are copied onto the third. This process continues until all reachable objects from the first memory pool have been relocated. A data structure is needed to keep track of intergenerational boundaries. After all relocation has taken place pointers from objects in older generations to younger relocated objects are updated to restore memory integrity.

Note that this algorithm automatically groups objects by lifespan. The average lifespan of an object is very short (local variables, literals) and most activity will take place in the youngest generation. Once objects are promoted to later generations the garbage collector only rarely has to reconsider them for deallocation. This is advantageous, since objects that have already survived several stages of garbage collection are likely to survive the next stage of garbage collection as well¹⁴.

A generational copying garbage collection strategy has implementation complexity and substantial memory overhead as its major downsides. Run-time efficiency, good locality and memory fragmentation avoidance are the positive attributes.

8.2.6 Conclusion Garbage Collection

A straightforward implementation of a mark-sweep garbage collection strategy is sufficient. The need for a better garbage collector will only arise for longer running programs that collect many objects. Since mark-compact and copying garbage collectors are substantially more complex to implement and still have substantial downsides. A generational copying garbage collector is in many ways preferable to a simplistic mark-sweep collector but for a prototype compiler the extra complexity cannot be justified.

8.3 Code Generation

In the prototype compiler of Chapter 5 and 6 the assembly code was generated with simple case analysis. Based on the x86 instruction reference a minimal amount of logic was used to emit machine code. For example: according to the x86 instruction specification a 32 bit value can be pushed onto the stack by with the by 0x50 followed by the four bytes that contain the literal. This works reasonably well for trivial instructions such as PUSH, but for more complex operations simple case analysis is not going to be sufficient. The MOV statement has 28 distinct encodings.

A solution is needed that allows the assembly-generating backend of the compiler to be written in as high-level a language as possible. Naturally, runtime performance is important: code is compiled and recompiled on demand.

A number of candidate backends have been considered: TCC, LLVM, NASM, GAS, libyasm, libjit, and LuaJIT. These assembly backends are evaluated in the following sections.

8.3.1 TCC

TCC[6] is the Tiny C compiler. It is completely ISOC99 compliant and capable of compiling a functioning Linux kernel. An attempt has been made to decouple the assembly-generating part from the rest of the compiler. This proved to be very challenging, as the compiler is approximately eighty thousand lines long and it uses an abstract intermediate layer for code generation, but there is no intermediate layer for the opcode generation itself! The assembly generating code is littered with constants that partially identify x86 instructions. For this reason TCC will not be a suitable candidate for assembly generation.

¹⁴The expected lifetime of an object is twice its age.

8.3.2 LLVM

The LLVM project[4] was started in 2000 as an intermediate language for compiler designers that can generate native code for a multiple different hardware architectures. It has optimization and transformation layers which allows the language designer to use a relatively high level intermediate language for communication with LLVM. The LLVM system is intended for and primarily used by statically compiled languages[17].

The Unladen Swallow project[3] is an attempt at a faster Python runtime. It is a branch of CPython and uses a JIT-compiler with LLVM as the backend. After several years of development of Unladen Swallow stopped; the project is now abandoned. One of the reasons given, by one of the developers was that LLVM in its current state generates efficient code, but does it slowly. Optimizations were also difficult in general because LLVM assumes languages with C-like attributes as input but for optimizations of a dynamic language high level knowledge of the language itself is needed.

The absence of successful JIT-compilers based on the LLVM system strongly indicates that LLVM is also unsuitable as a code-generating backend.

8.3.3 NASM, GAS

Two open source x86 assemblers have been considered for use as backend: NASM[22] and GAS[1]. Conceptually the use of an existing x86 assembler is simple: compile the code that is to be just-in-time compiled into a sequence of x86 assembly statements and feed this into one of the assembler packages. Load the result into memory, and continue from there.

There are some downsides to this approach, however. Connecting an assembler to the compiler via a pipe is fragile. There is no good way to read error messages from the assembler, since they are in a human-readable format. Additionally, assemblers assume the input is well formed: input files are either complete programs or libraries. For JIT purposes often only a few specific instructions need to be compiled, and the command line interface of open source assemblers do not afford this functionality.

So to create a reliable interface between the assembler and the rest of the JIT-compiler the command-line interface has to be replaced with a compiler-to-compiler interface. Due to the complexity and archaic designs of these assemblers this will be seen as a last resort.

8.3.4 Libyasm

The `libyasm` library is part of YASM[15] a portable and modular rewrite of the NASM assembler. Libyasm is the main interface to NASM and is meant to be integrated with compilers and debuggers. Unfortunately, no JIT compilers (or any compilers, for that matter) could be found that used libyasm as backend. The documentation seems to be incomplete and not a single research paper that refers to libyasm could be found. Even though it initially looked promising it turns out that libyasm is not (yet) suitable as a backend for the JIT compiler.

8.3.5 libjit

The libjit[2] system is a GNU library for run-time compilation. It consists of a jit-compiler backend and a virtual machine that implement a low level instruction set. Unfortunately,

libjit defines how the CPU registers are used and how variables and objects are stored in memory. This means libjit cannot be used as a simple x86 JIT-assembler.

8.3.6 LuaJIT

LuaJIT is a Just-In-Time compiler for Lua, written by Micheal Pall. The LuaJIT compiler comprises only 15 thousand lines of C code¹⁵ making this the smallest JIT-compiler by a significant margin.

The LuaJIT compiler has a separate assembly code generator called DynASM written in Lua. DynASM extends the C language with notation for runtime generated assembly code. When compiling the JIT-compiler the DynASM preprocessor will take the assembly fragments from the C code, assembles it and places the precompiled assembly fragment as a constant byte array into the C file. This is a perfect fit for general JIT-compiler construction. In the next section the DynASM system will be covered in detail.

8.4 DynASM

DynASM translates assembly snippets into precompiled x86 code. When the JIT-compiler runs the x86 opcode is constructed by locating and concatenating the right precompiled snippets and inserting the correct memory addresses and values, which depend on runtime information.

For instance, the short shorthand assembly statement:

```
mov [ebp - 4], eax
```

is equivalent to the GNU assembler statement:

```
mov eax, dword ptr [ebp-4]
```

and is interpreted as: subtract 4 from the `ebp` register, dereference the value and assign the dereferenced value to the `eax` register. It is a 32bit `mov` instruction with a register and offset as source and a register as destination. There are over twenty different variations of `mov`: `mov` instructions for direct and indirect memory access, for different register sizes (byte, short, long) and for different kinds of offsets. In this case the appropriate version is:

```
MOV r/m32,reg32 ; o32 89 /r [386] ; [ece390 illinois.edu]
```

A move instruction in 32 bits mode (`o32`) that is supported on at least 386-class processors. It uses a `modr/m` (mod register/memory) byte to encode the form of the arguments. The `/r` part indicates that the register field in the `modr/m` field is used to encode the destination. This version of `mov` starts with the byte `0x89`.

The `modr/m` byte consists of three parts. The first 2 bits indicate whether displacement bytes are used (0, 1, or 4 bytes displacement in 32 bit mode). Since we want to subtract 4 we only need a single byte displacement. Bits 3 to 5 indicate the register of the destination. Since the target register is `eax` (the 0th register: 000) these three bits can stay zero. The final 3 bits are used for the source register which is `ebp` (the 5th register: 101). So the `modr/m` byte becomes `10 000 101 = 0x45`.

¹⁵LuaJIT-1.1.5, as measured by SLOCCOUNT

The `modr/m` byte indicated a 1 byte displacement is needed. The integer `-4` is `0xfc` in two's complement 8 bit representation. So the final representation of the `mov` instruction is `0x89 0x45 0xfc`. This should illustrate that the case-based approach as used in Chapter 5 is insufficient for generating more complex assembly code and that an intermediate assembly layer as provided by DynASM is really necessary.

The DynASM system can best be illustrated using another example. The switch statement below partially implements some instructions:

Listing 9: Compiler with inline assembly

```

2      switch (instruction.op) {
        case PUSH:
            | push eax
4          break;
        case POP:
            | pop eax
6          break;
8          case ADD.C:
            | add eax, instruction.value
10         break;
        ...
12     }

```

The lines that start with a pipe character (`|`) are interpreted by the preprocessor. The DynASM preprocessor will then output the following C code:

Listing 10: Compiler with compiled assembly

```

2      switch (instruction.op) {
        case PUSH:
            // | push eax
4          dasm_put(Dst, 540);
            break;
6          case POP:
            // | pop eax
8          dasm_put(Dst, 751);
            break;
10         case ADD.C:
            // | add eax, instruction.value
12         dasm_put(Dst, 212, instruction.value);
            break;
14         ...
        }

```

And the raw assembly is added to the C source file. With all source files in place the JIT-compiler can now be compiled. When code in the dynamic language is evaluated the dynamic language compiler is invoked and the dynamic language statements are compiled into bytecode. This bytecode is then passed on to the JIT-compiler that performs the following tasks:

1. Concatenate the pre-compiled assembly fragments in the sequence of the program bytecode.

2. Replace the variable parts of the assembly fragments with the correct addresses and values.
3. Calculate and fix all the relative jump addresses.

The generated code can now be executed. As execution continues the dynamic program will run into a function or method that has not yet been compiled and the JIT-compiler will interrupt execution and compile the necessary functions and methods.

8.5 Register Usage

The x86 machine architecture has eight full-size registers: **EAX** Accumulator, **EBX** Base Register, **ECX** Source Register, **EDX** Data Register, **ESI** Source Index, **EDI** Destination Pointer, **EBP** Base Pointer, and **ESP** Stack Pointer.

For the JIT compiler one register is needed as accumulator, as described in section 5.4. For this **EAX** register is the natural choice. The **EBX** will be a general purpose register that can be used during for the calculation of intermediate values. The **EBP** register is used as the base pointer. Local variables and function arguments have fixed offsets from the base pointer, and as such can be accessed efficiently. The stack pointer **ESP** is used in the conventional manner. The **EDI** register is the free pointer and is used for memory allocation. The **ECX** and **EDX** registers are used as intermediate registers and may be freely overwritten by any instruction. The remaining registers have no fixed function and can be freely used by the JIT-compiler.

By assigning a fixed function to a number of the registers the assembly code generated by the JIT-backend becomes substantially less complex. To illustrate, consider the **DUP** instruction that duplicates the top of the stack. Given that the **EDX** register cannot contain any important data the trivial implementation in two instructions is guaranteed to be correct:

```
mov edx, [esp]
push edx
```

Without the convention that the **EDX** register may be used freely the burden is now on the **DUP** instruction (a) prove that the **EDX** register happens to be unused in the current context (difficult) or (b) that the value in **EDX** is not altered by the instruction (inefficient). The assembly code for the **DUP** instruction would then have to take the following inefficient form:

```
mov [buf_addr], edx
mov edx, [esp]
push edx
mov edx, [buf_addr]
```

Where `buf_addr` denotes a memory slot that has been allocated ahead of time that can be used for temporary storage.

Between functions calls the values of all registers except for **EAX**, **ECX** and **EDX** are preserved, conform the `cdecl` calling convention. By honoring this convention the language environment can seamlessly jump between C library functions and JIT-compiled functions.

8.6 Pass by value

As usual for dynamic languages, assignment is *by value*, meaning that a value is copied on assignment. Objects are created on the heap and variables contain references to objects; not the objects themselves. On assignment the reference is copied, conform *by value* semantics.

8.7 Local Variables

In a dynamic language objects are usually allocated from heap memory that is governed by the garbage collector. This is necessary since the lifetime of an object is defined by its reachability: once an object can no longer be reached it is considered garbage and it will eventually get deleted by the garbage collector.

If the JIT-compiler can recognize a variable is used strictly locally then a number of optimizations can be made:

- the object does not need to be boxed or encoded,
- the object can live on the stack; the garbage collector is not needed, and
- when the object goes out of scope the memory can be re-used immediately.

In section 5 we showed that these optimizations are significant using the prototype compiler and virtual machine. After all, a straightforward interpreter or bytecode VM has to implement even an operation as simple as the addition between two floating point values by allocating a new floating point object to store the result. In general: every mutable operation (except assignment) on an object requires the allocation of a new object.

A JIT-compiler cannot determine when a variable is local (except for a few select cases). If an innocuous operation, such as passing a variable to a function, may affect the lifetime of the variable: a reference to the variable may be kept in memory.

The problem of unnecessary allocations is partially circumvented by storing small integer as encoded pointers (section 6.3). The problem remains for composite types and the other primitive types (floating point values, strings, booleans). Notably, the primitive objects all fit into a single register. So by unboxing¹⁶ the primitive objects the unnecessary allocations are prevented. This introduces a new problem, when the primitives leave the local scope they require their type tag again, which is no longer known. To solve this problem the concept of a *Type Stack* is introduced.

0x000002A	INTEGER
0x0000001	BOOLEAN
0xAD4EFED	STRING
Frame Pointer	NONE
Return Address	NONE
Function Object	OBJECT
Fun Argument 1	Type Arg 1
⋮	⋮
Fun Argument <i>N</i>	Type Arg <i>N</i>

Figure 4: Stack and Type Stack side-by-side

¹⁶Stripping the type tag from the object

A *Type Stack* is a stack parallel to the ordinary stack that contains the type tags that have been stripped from the primitive objects. A set of *type registers* are introduced that are parallel to the ordinary registers. As an invariant, a type register always contains the type of the object of the corresponding register. When an object is pushed onto the stack the type of the object is pushed onto the type stack. When a literal is assigned to a register the type of the literal is assigned to the corresponding type register. Figure 3 shows the type stack side by side the conventional stack. The stack has the conventional layout: function arguments followed by the stack frame followed by the local variables and temporary storage.

When a primitive variable leaves the local scope the type stack is used to allocate a new object of the correct type. Primitive objects can remain unboxed in during function calls and method invocations. The Type Stack approach greatly reduces the frequency at which new objects are allocated at small price: the cost of unboxing and boxing the primitive values and the cost of maintaining integrity of the type stack.

8.8 Conclusion

In this chapter the different design considerations have been covered. A garbage collection algorithm has been chosen. A comparison of JIT-libraries and assemblers lead us to conclude that LuaJIT was the most suitable for general-purpose JIT-compiler construction.

The conventions for register usage and function calls have been covered. A technique that reduces the number of unnecessary memory allocations has been covered.

The remaining design decisions, such as language syntax, library functions, and the object system are completely standard for a generic dynamic language and are not described in detail.

9 Conclusion

9.1 Results

The research in Dynamic Language Performance with JIT-compilation yielded the following results (in order of appearance):

1. We have shown that a very primitive virtual machine has virtually identical performance characteristics as CPython for some CPU-bound algorithms (chapter 5).
2. A primitive JIT-compiler has been constructed in less than 100 lines of code. Performance results demonstrate that the replacing a virtual machine with the simplest of JIT-compiler can immediately give significant benefits. We have shown that for some CPU-bound algorithms the primitive JIT-compiler comes with 50% of the performance of the equivalent C program (chapter 5).
3. We demonstrated that a type-inference based approach is not feasible for dynamic languages because no type inference algorithm can correctly identify co-variant and contra-variant types (chapter 6). We emphasize that because type tests are so inexpensive only very few resources can be spent on detecting unnecessary tests.
4. We have determined that the cost of encoding integers (chapter 6) is free. The intuitive assumption is that repeatedly decoding and encoding integers is going to lead to a measurable decrease in performance, but this turns out to be not the case. Based on our measurements and the on related research we conclude that elimination of overflow tests in a purely dynamic environment are unlikely to ever produce substantial performance gains.
5. We evaluated a dozen different assembler backends (chapter 8) and concluded that all were unsuitable for embedding in a JIT-compiler, except for LuaJIT which turned out to be perfect for JIT-compilation and general purpose generation of assembly code. The LuaJIT assembler was moderately easy to separate from the rest of the LuaJIT compiler.
6. A Type Stack was introduced as an technique for unboxing primitive local variables (chapter 8). By keeping track of type information in a secondary stack unnecessary allocations are prevented (and therefore fewer garbage collection cycles are needed). No similar technique has been found in the literature.

From the research as a whole we conclude that the creation of prototype virtual machines and prototype JIT-compilers is an effective way to measure the performance characteristics of dynamic programming languages. We also conclude that the performance gains offered by JIT-compilation are significant, even if only basic JIT-compilation strategies are employed.

9.2 Future Work

In this research project the JIT-compiler and the interpreters have deliberately kept simple and as such some optimizations have not been made. One possible direction for future work is to investigate the performance gains that can be made with more advanced classical compiler optimizations. For instance, graph based register allocation algorithms can be used, or

common subexpression elimination. Another venue for optimization is function inlining or by rewriting tail-recursive functions into the procedural equivalent.

More performance gains may be achieved by adding a second bytecode layer to the compiler. A high level bytecode layer allows for high level structural optimizations, whereas a low level bytecode layer allows for many peephole optimizations. We estimate that by adding second bytecode layer and some simple peephole optimizations the length of the generated assembly can be reduced by a third.

Finally, trace-based JIT optimization techniques can be applied. Heuristics and measurements are used to determine which parts of the program are executed frequently. Program traces of these sections can then be recorded. These traces contain types information and other useful assumptions for further optimization. Based on these the trace information specialized (and more efficient) code can then be created. Because there are only few hot code paths in a program more aggressive optimization techniques can be applied.

9.3 Reflection

During the course of this project many different areas of research were touched upon. With hindsight one detour probably could have been avoided: a problem with type inference lead into a detour into type theory and ultimately let us to conclude that one of the underlying assumptions of our approach was flawed. A better study of the literature beforehand would ultimately have been better.

Aside from this issue the research has yielded multiple interesting results. Often we had the satisfaction of finding our intuitions confirmed by the prototypes or by related research projects. In some cases the findings did not line up with our intuitions and we changed course accordingly.

Based on the results enumerated above we are pleased to conclude that the research has yielded multiple interesting results. During the research we came across numerous research projects that unsuccessfully attempted to improve the performance characteristics of established dynamic languages. Some of those research projects made assumptions about performance characteristics that turned out to be incorrect (section 4.2). This paper may perhaps, amongst other things, persuade researchers that small-scale prototypes can accurately model the performance characteristics of real-world language environments.

9.4 Acknowledgments

First and foremost I would like to thank my supervisor Mark van den Brand for his excellent guidance and help. My gratitude also goes to Mike Pall, the architect and programmer of LuaJIT. Finally, my thanks go open source contributors across the globe, without whom research like this would not be possible in the first place.

References

- [1] Gnu assembler, part of gnu binutils.
<http://www.gnu.org/s/binutils/>.
- [2] Library for just-in-time compilation for dotgnu.
<http://www.gnu.org/software/dotgnu/libjit-doc/libjit.html>.
- [3] Unladen swallow: A faster implementation of python.
<http://code.google.com/p/unladen-swallow/>.
- [4] Vikram Adve and Chris Lattner. Low level virtual machine (llvm).
<http://llvm.org>.
- [5] Lars Bak. Chrome v8 javascript engine.
code.google.com/apis/v8/design.html.
- [6] Fabrice Bellard. Tiny c compiler.
<http://bellard.org/tcc>.
- [7] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.
- [8] Brett Cannon. Localized type inference of atomic types in python, 2005. Master Thesis at California Polytechnic State University, San Luis Obispo.
- [9] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. In *Technical Report No. 07-10, Donald Bren School of Information and Computer Science, University of California, Irvine*, sep 2007.
- [10] Karel Driesen. Software and hardware techniques for efficient polymorphic calls. Technical report, Santa Barbara, CA, USA, 1999.
- [11] Michael Furr, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for ruby. In *ACM Symposium on Applied Computing*, pages 1859–1866, 2009.
- [12] 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. *SIGPLAN Not.*, 44:465–478, June 2009.
- [13] Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, jul 2005.
- [14] Daniel Ingalls. The smalltalk-76 programming system. In *POPL*, pages 9–16, 1978.
- [15] Peter Johnson. The yasm modular assembler project.
<http://yasm.tortall.net/>.
- [16] Richard Jones and Rafael Lins. Garbage collection: Algorithms for automatic dynamic memory management. Technical report, 1996.

- [17] Chris Lattner, Vikram Adve, et al. Llvm: A compilation framework for lifelong program analysis and transformation. In *In proceedings of the international symposium on code generation and optimization*, 2004.
- [18] Mozilla. The mozilla spidermonkey javascript engine.
www.mozilla.org/js/spidermonkey/.
- [19] Michael Salib. Faster than c: Static type inference with starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.
- [20] Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software, CC'11/ETAPS'11*, pages 2–21, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19860-1.
- [21] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 280–295, 2004.
- [22] Simon Tatham and Julian Hall. The netwide assembler.
<http://www.nasm.us/>.
- [23] Kresten Krab Thorup. Optimizing message lookup in dynamic object-oriented languages with sparse arrays. (Publisher unknown).
- [24] Webkit. Squirrelfish javascript interpreter for webkit.
trac.webkit.org/wiki/SquirrelFish.
- [25] Salikh S. Zakirov, Shigeru Chiba, and Etsuya Shibayama. Optimizing dynamic dispatch with fine-grained state tracking. In *Proceedings of the 6th symposium on Dynamic languages, DLS '10*, pages 15–26, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4.