A JAVA VIRTUAL MACHINE EXTENDED TO RUN PARROT BYTECODE

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2006

By
Martin Dahl
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Parrot is a virtual machine designed for dynamically typed languages. Originally concieved as the runtime system for Perl 6, Parrot has taken inspiration from a multitude of languages, such as Python, Ruby and Scheme. This thesis presents an extension to the Jikes Research Virtual Machine to make it run Parrot bytecode. Targeting the Jikes RVM optimizing compiler, good performance is expected once the program matures.

In addition to the JVM extension, the native class library available to Parrot programs is also ported to pure Java. This should make for an improvement on the C-implemented version in Parrot, clearing up some of the muddled inheritance hierarchy.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to give thanks to my supervisor, Prof. Ian Watson, for letting me do such a fun and challenging project, and to Dr. Ian Rogers, whose overflowing enthusiasm and optimism always leaves me rearing to go back at it.

Also, to both Jikes RVM, Parrot and Perl 6 teams, without whom this project could never have happened.

Chapter 1

Introduction

JVMs are often extensible and capable beyond being just JVMs. This project will extend a JVM to make it a Parrot VM. By reusing the optimizing compiler of the JVM, good performance should follow.
The initial project offering.

1.1 Objectives and Motivation

As stated in the quote above, the tangible objective of the project is an extension of a Java virtual machine to run Parrot [Par] bytecode. Initially, as reflected in the title given in the list of project offerings, *"Java Perl"*, the hope was to be able to run Perl 6 code on top of this extended JVM. As time passed, this part of the objectives fell away more and more, mainly due to the incompleteness of Perl 6 itself at this point.

The JVM extended by the project is the Jikes RVM [AAB00, RVM]. This is partially because it is the development JVM preferred by the Jamaica project [JAM], but also for other reasons described in more detail in a later chapter.

Some of the more important objectives for the project, then, are as follows:

- provide an as complete as possible implementation of Parrot on top of the Jikes RVM,
- utilise the JVMs optimizing compiler to obtain good performance (hopefully surpassing even Parrot itself),
- and, perhaps, contribute somewhat to making the Jikes RVM a little less Java-centric.

1.2 Related Work

This section will present a few projects that are in some way or another related to this one.

1.2.1 Parrot

First and foremost, of course, is Parrot itself. Described in detail in chapter 2, the Parrot virtual machine is what this project will be duplicating. As will be seen, Parrot consists of multiple parts, some of which are more relevant to this project than others – mainly the virtual machine and the class library are considered interesting at this point, though once these parts of the project mature, effort will likely be put into others.

Parrot grew out of the Perl 6 project, as the intended target platform for the Perl 6 compiler. However, at the time Parrot was started, the language specification was far from complete, so while Parrot was concieved for Perl 6, it has been influenced by many other languages, and is currently seen more as a general virtual machine designed, though designed mostly with dynamically typed languages in mind.

1.2.2 Pugs

Pugs [PUG] is probably the most complete Perl 6 implementation at the time of writing. Starting as an academic exercise of implementing a subset of the Perl 6 language in Haskell, it rapidly grew beyond all expectations. It now tracks the progress of the Perl 6 specifications closely.

Pugs is designed to enable the use of multiple back-ends. It can use Parrot itself for the actual execution of code, or even generate and output several of Parrots assembly language formats. It does tend to create a huge amount of code for the simplest programs though, and so has not been as useful as might have been hoped in testing Parakeet so far. Hopefully, as Parakeet matures, that will change, making Pugs very useful indeed for the furthering of the project.

1.2.3 PearColator

While PearColator [Mat04], a dynamic binary translator, is not directly related to Parakeet or the Parrot virtual machine, it bears mentioning here because it (ab)uses the Jikes RVM in much the same fashion as Parakeet does.

PearColator extends the Jikes RVM to run, in place of Java bytecode, PowerPC binaries. Being able to study how PearColator works as an extension of the RVM has been a great help, especially at the beginning of the project.

1.3 Organisation of the Thesis

Chapter 2 describes the Parrot project. The chapter starts by detailing the history of Parrot, from its conception in the Perl 6 internals project, to where it stands today. It then goes on on discussing the various parts of the project and the design of the software.

Chapter 3 gets down and technical with Parrots bytecode format. Most all aspects are described in detail. This bytecode is what Parakeet (the software created for this project) is designed to run.

Chapter 4 is about the project itself. Mainly how the software is designed, the various pieces that make it up and how they fit together. There is also a brief discourse on how the project has been conducted and some of the problems that have been run into underways.

Chapter 5 describes performance benchmarking performed on Parakeet. The results from the benchmarks are displayed, and some explanation is attempted.

Chapter 6 wraps up the thesis. It summarises the project and discusses some possible (and necessary, if Parakeet is to amount to anything beyond this thesis) future developments.

Chapter 2

The Parrot Project

> *"Parrot is a virtual machine designed to efficiently compile and execute bytecode for interpreted languages. Parrot will be the target for the final Perl 6 compiler, and is already usable as a backend for Pugs, as well as variety of other languages.*
>
> *Parrot is not about parrots."*
>
> **Definition from http://parrotcode.org/**

2.1 Perl 6

This chapter will introduce and describe the Parrot project in some detail. Such a discourse should start with the historical roots; Parrot was first concieved as the target virtual machine for the Perl 6 language. This section, then, deals with how Perl 6 came into being, its design process and the progression to its current state.

2.1.1 Design

Perl 6 is a complete redefinition of the Perl 5 language. It keeps many features from Perl 5, but also removes some and introduces new ones in an incompatible way. The whole process on the new revision started in earnest mid-2000, first announced by Larry Wall in his *State of the Onion 2000* [Wal00]. In his speech on the same subject at the fourth annual Perl conference, he is quoted as saying *"Perl 5 was my rewrite of Perl. I want Perl 6 to be the community's rewrite of Perl and of the community."*

So the design process was started. Members of the community submitted formal suggestions in the form of *request for comments* (RFC). More than 350 such RFCs were proposed in this initial phase. These were then reviewed and discussed, resulting, over the span of the next few years, in three sets of documents:

- The *Apocalypses*, authored by Larry Wall, represent his thoughts on the individual RFCs and act as a starting point for the design of the new language.
- The *Exegeses*, written by Damian Conway, another prominent Perl hacker, comment on the Apocalypses and expand and explain through examples how the various changes will affect the language.

- Finally, the *Synopses*, which are mostly summaries of the Apocalypses and Exegeses, condensing these into more formal design documents.

2.1.2 Perl 6 Internals

The second part of the Perl 6 project is the *Perl 6 internals*. This deals with actually implementing the design. Around mid-2001, this resulted in the Parrot subproject being started. The goal was to provide a language-neutral runtime environment. As a short aside, the name "Parrot" comes from an April Fools' joke by Simon Cozens [Coz01], presented as an interview with Larry Wall and Guido van Rossum (the creator of the Python language), detailing plans to merge Perl and Python into a new language called Parrot.

All features of dynamic languages would be catered to, it was decided, and threading and Unicode support would be built in from the start. The two latter had been some of the most problematic features to add to Perl 5, and everyone wanted to avoid such problems this time around. At this point, of course, the design for Perl 6 was far from finalised, and this separation of implementation from the actual syntax of the language was benificial – Parrot development could begin even though Perl 6 was still very much a moving target.

Originally a side-effect, though later recognised as a huge benifit, was the fact that since Parrot was not tied to Perl 6, and in fact incorporated features from other languages such as Ruby, Python and Scheme, Parrot could act as a target runtime environment for many other languages, even to the point where libraries written in one language could easily be called from a program written in another. Parrot allows for picking and choosing languages, each for its best fit, and as languages are usually better suited to some task than others, this can be a huge boon. This is not unlike the .NET *Common Language Runtime* [MG00], though the .NET VM did not exist when the Parrot project was started, or, as stated in the Parrot FAQ, *at least we didn't know about it when we were working on the design*

2.1.3 Pugs

Pugs, the Perl6 User's Golfing System [PUG], is an implementation of Perl 6, as described in the Synopses. Pugs was started as an academic exercise by Autrijus Tang in February 2005, to implement a purely functional, reduced set of Perl 6 in Haskell. As stated in a March 2005 interview [chr05], the "academic exercise" part went out the window after about two days, the "purely functional, reduced set" following the next day.

Pugs is, at the time of writing, the most complete implementation of Perl 6. This is mostly because the Pugs developers religiously track changes to the Synopses, and any change to a Synopse that causes Pugs to be incompatible is considered a bug (in Pugs). The project provides valuable feedback to the Perl 6 design, both as a working implementation that can be tinkered with and as a sanity check for design changes (as in many cases it can be notoriously difficult to understand what a change would really do before it is actually implemented).

It is also the intention of the Pugs developers that Pugs will help with bootstrapping the final Perl 6 compiler. The goal is to have a self-hosting Perl 6, with the compiler itself being written in Perl 6. Obviously this is a chicken and egg problem, the compiler being unable to compile itself until it has been compiled. Pugs, as a working Perl 6 implementation, can help by providing that first compiler.

2.1.4 Parrot Perl 6 Compiler

The Parrot project ships with a compiler for Perl 6. This compiler generates *Parrot Intermediate Representation* (PIR) code from Perl 6 programs. PIR is discussed further in section 2.2.5. Unfortunately, this compiler is rather incomplete at the time of writing.

This compiler uses the *Parrot Grammar Engine* (PGE). A conglomeration of regular expressions, Perl 6 rules, a grammar engine and a parser, PGE supports many of the advanced Perl 6 regular expression and parsing features, mixed in with some appropriated from Perl 5. It also includes a grammar compiler, to convert entire grammar specifications into PIR. Both the Perl 6 compiler and PGE are themselved implemented entirely as PIR code.

2.2 Virtual Machine

Now that some of the history behind the Parrot project has been discussed, the time has come to move on to the more technical details of the virtual machine itself. As mentioned already, Parrot was designed from the beginning with dynamically typed languages in mind, to run programs written in such programs more efficiently than virtual machines designed for statically typed languages (like the JVM or .NET) would be capable of.

2.2.1 Design

Three main principles drive the design of Parrot – speed, abstraction and stability [RST04, chapter 8]. Stated briefly, these are more or less the following:

- First, *speed*, is of the utmost importance. If Parrot is slow, any program, no matter how well designed, will not be able to get good performance. Parrots efficiency sets the upper bound on how well programs written for it can run. This principle, though, is not only about execution time. Resource usage also comes into play. If Parrot requires too much memory for instance, it might overflow into disk-backed memory and the system could start thrashing, or the virtual machine might get killed by the operating system for running the machine out of memory. Good performance would clearly not ensue. Obviously, some balance is required between raw execution speed and resource usage.

- On to *abstraction*. This point is important because Parrot is a large system, and it would be nigh impossible for anyone to know everything about all parts, in detail. Good abstraction allows one to know some parts only on a general level, so long as there is trust that the underlying details will not break the abstraction.

- Last, but not least, *stability*. Parrot will provide a backend for many languages. To enable this, and be able to execute programs, even after substantial amounts of time has passed, Parrots interfaces must remain stable. Also, Parrot is meant to be embeddable into other programs, so a stable interface on the binary level, for linking, is required as well.

The actual design is a balance between these. Speed is often seen as the most inportant of the three, and this does at times lead to compromises being made, with the other two suffering to some degree.

2.2.2 Architecture

Parrot is divided, more or less, into four distinct main parts. These are the *parser*, the *compiler*, the *optimizer* and the *interpreter*.

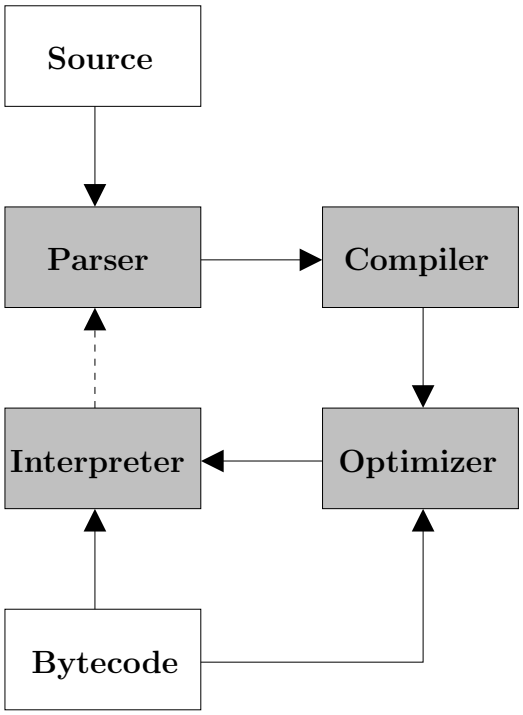

Figure 2.1: The flow through Parrots main parts

Between these parts, control flows as shown in figure 2.1. Programs can enter the system in two ways, either as source code or as previously generated bytecode.

- *Source code* is parsed into an *Abstract Syntax Tree* (AST). This AST is passed to the compiler, which turns it into bytecode that can be executed by the interpreter. This code, along with the AST, is then handed over to the optimizer. Based on the parameters given, the bytecode is optimized either lightly or heavily (in some cases heavy optimization is a waste of time, for instance if the optimization phase would take longer than the runtime of the unoptimized program) and finally sent on to the interpreter which

executes it. Depending on the program, the interpreter may call back into the parser, and the whole process starts over again.

- With *bytecode*, the first two phases are skipped, going straight to the optimization, or indeed, again based in parameters given, directly to interpretation – if the bytecode was previously heavily optimized, there is little sense in going through the motions again when there is nothing to gain.

2.2.3 Registers

The core design of Parrot is akin to that of a register-rich CISC CPU, though it also bears some resemblance to modern RISC CPUs; for instance, all operations are performed on data in registers. Using such a design, the Parrot team can draw on decades of research into compilers, as most such research the last 30+ years has been on register systems of one sort of another. Parrot has four types of registers: integers, numbers (floating point), strings and PMCs (see section 2.3). There are 32 registers of each type available.

This register-based architecture goes somewhat against the grain of current trends in virtual machines, which are nowadays most often stack-based. Though studies like the 2005 one by Shi, et al. [SGBE05] have shown that a register-based virtual machine can reduce runtime on the order of 30% for standard benchmarks, the Parrot team appears not to have had such lofty goals, as the following quote from the Parrot FAQ shows:

> *"**What's with the whole register thing machine?***
>
> *Not much, why do you ask?*
>
> ***Don't you know that stack machines are the way to go in software?***
>
> *No, in fact, I don't.*
>
> ***But look at all the successful stack-based VMs!***
>
> *Like what? There's just the JVM.*
>
> ***What about all the others?***
>
> *What others? That's it, unless you count Perl, Python, or Ruby.*
>
> ***Yeah them!***

> *Yeah, right. You never thought of them as VMs, admit it.* :^)
>
> *Seriously, we're already running with a faster opcode dispatch than any of them are, and having registers just decreases the amount of stack thrash we get.*
>
> ***Right, smarty. Then name a successful register-based VM!***
>
> *The 68K emulator Apple ships with all its PPC-enabled versions of Mac OS.*
>
> ***Really?***
>
> *Really."*

[RST04, pp. 106] also chimes in on the subject with: *It's also just more pleasant for us assembly old-timers to write code for.* Other examples of successful register-based virtual machines are the Jikes RVM, and of course, by now, Parrot itself.

2.2.4 Instruction Set

Parrot has a multitude of different instructions, ranging from core ones like *branch* or *end* through bitwise operations, string operations, operations for I/O and for manipulating objects to obscure trigonometric operations. In addition to this large number of instructions, each one can have one or more opcodes defined, varying with the various parameters the opcode takes.

A good example is the *substr* instruction. There are 27 possible permutations (with 27 corresponding opcodes) for this instructions, depending on whether a length is provided, whether the start or length parameters are integer literals or come from registers, and even if an optional replacement for the substring is specified. Obviously such a system results in a huge number of opcodes, counting in at more than 1200 in the 0.4.5 release of Parrot.

In addition to all these opcodes released with the virtual machine itself, opcode libraries can be loaded dynamically within Parrot. This allows for high-level language compilers to use their own opcodes, which can in some cases be more efficient than expanding the Parrot standard library (see section 2.3 for more about the standard library).

2.2.5 Parrot Assembly Language and Intermediate Representation

Parrot Assembly (PASM) is an assembly language created for the Parrot virtual CPU. It is mostly just a set of mnemonics for the various instructions, plus some common assembly language features like labels for branch and jump targets. Owning to the dynamic language support in Parrot, many rather un-assembly-like features are supported, like objects, garbage collection and more. See figure 2.2 for an example of PASM code.

Parrot Intermediate Representation (PIR) provides an (albeit shallow) abstraction on top of PASM. It allows for code to be organized into subroutines, with easily defined parameters and return values. PIR also adds macros, both temporary and named registers, and a whole host of syntactical sugar. In all, it provides a much nicer environment for the programmer, and is now also considered the prime target for compilers targeting Parrot. See figure 2.3 for an example of PIR code (this is the same program as in the PASM example).

Both code examples are taken from the Parrot distribution.

2.3 Parrot Magic Cookies

Parrot Magic Cookies (PMCs) are Parrots way of supporting complex types. Anything beyond that provided directly by registers (integers, floating point numbers and strings) must be implemented as a PMC. These are separate from the virtual machine, though the VM does use some core PMCs internally. PMCs can be implemented either in C or in PIR / PASM.

2.3.1 Core PMCs

Parrot comes with a core set of PMCs. These are automatically loaded on startup, and are available to all code written for Parrot. Mostly, the core PMCs are fairly mundane constructs, like classes for integers, strings, arrays and so on. These are all written in C for efficiency reasons.

```
# I0 = last
# I1 is the counter for the loop
# N2 is the argument for gen_random
# N3 is the return from gen_random
main:
    get_params "(0)", P0
    elements I0, P0
    eq I0, 2, hasargs
    set I1, 900000
    branch argsdone
hasargs:
    set S0, P0[1]
    set I1, S0
argsdone:
    set I0, 42
    unless I1, ex
    set N2, 100.0
while_1:
    bsr gen_random
    dec I1
    if I1, while_1
    new P0, .FixedFloatArray
    set P0, 1
    set P0[0], N3
    sprintf S0, "%.9f\n", P0
    print S0
ex:
    end

.constant IM 139968
.constant IA 3877
.constant IC 29573

gen_random:
    mul I0, .IA
    add I0, .IC
    mod I0, .IM
    set N1, I0
    mul N3, N2, N1
    div N3, .IM
    ret
```

Figure 2.2: Example PASM Code

```
.sub main :main
    .param pmc argv
    $S0 = argv[1]
    $I0 = $S0
while_1:
    gen_random(100.0)
    dec $I0
    if $I0 > 1 goto while_1
    $N0 = gen_random(100.0)
    $P0 = new .FixedFloatArray
    $P0 = 1
    $P0[0] = $N0
    $S0 = sprintf "%.9f\n", $P0
    print $S0
    .return(0)
.end

.const float IM = 139968.0
.const float IA = 3877.0
.const float IC = 29573.0

.sub gen_random
    .param float max
    .local float last
    last = 42.0
loop:
    $N0 = last
    $N0 *= IA
    $N0 += IC
    $N0 %= IM
    $N1 = max
    $N1 *= $N0
    $N1 /= IM
    last = $N0
    .yield($N1)
    get_params "(0)", max
    goto loop
.end
```

Figure 2.3: Example PIR Code

The core PMCs form the stem and first few branches of the PMC inheritance tree. PMCs all inherit from the *default* PMC, which specifies a host of methods that can be overridden. The PMC system also allows for mix-in style inheritance, where the developer can specify things like: "use `Array.get_string()` as this PMCs `get_string()` method".

2.3.2 Other PMCs

Magic cookies not shipped with Parrot can be dynamically loaded or even created at runtime, and there is a whole set of instructions specifically to create and modify classes (these also work on core PMCs – the developer is free to replace the `get_integer()` method in the Integer class if she so chooses, though doing so would likely provide for much confusion and head-scratching some time down the line when something has gone wrong and needs to be debugged!).

High-level language support modules can also supply their own versions of core classes. For instance, the Perl 6 infrastructure provides PerlArray, PerlString and so on to support all the features Perl 6 requires from those classes.

2.4 Compiler suite

Finally, the last section of this chapter will look briefly at compilers shipped with Parrot. Compilers for a multitude of languages are provided, and there is also a more generic compiler framework that compiler writers can use to simplify the construction of new compilers.

2.4.1 Perl 6

The Perl 6 compiler has been discussed already in section 2.1.4.

2.4.2 Other languages

Parrot ships with compilers for a host of languages, in varying state of completeness. Unsurprisingly, most of these are for dynamically typed languages, like Python, Ruby, Scheme or Lua. There are also compilers for more esoteric languages, like Ook or Befunge, probably because these languages are, while usually

turing complete, very small and simple (from the compiler standpoint that is – some of them are notoriously write-only form a programmer point of view).

2.5 Summary

- Parrot was born from the Perl 6 project, meant to be the final target for the Perl 6 compiler. It is meant to be *the* virtual machine for dynamically typed languages.
- A Perl 6 compiler is shipped with Parrot, but it is rather rudimentary at this stage. The most complete Perl 6 implementation at the moment is Pugs.
- Parrot is designed for speed. Also, abstraction in the code and stability of interfaces are important points in its design.
- There is a flow of control between the four main parts of Parrot. These parts are the parser, the compiler, the optimizer and the interpreter. The flow is from left to right as they are written here, though the interpreter can call back into the parser, at which point the process starts all over again.
- Parrot is register based rather than stack based. This may go against "common wisdom", but there are good reasons, both technological and otherwise, for taking this path.
- The number of opcodes in Parrot is large, mainly because each instruction can map to many opcodes depending on the various parameters the instruction can take, but also because there is a trend to make instructions out of common (and at times, not so common) operations.
- There are two main languages for writing programs for Parrot, PASM and PIR. PIR provides some higher-level constructs, and is generally much nicer to work with.
- Parrot comes with a library of classes for anything that requires complex types. These classes can be loaded at run-time, and the library is user-expandable.

- Finally, Parrot also comes with a compiler suite, geared towards generating PIR from specific language source code.

Chapter 3

Parrot Bytecode Format

This chapter will present the Parrot bytecode format in some detail. Parrot stores its bytecode on disk in a format called PackFile, which consists of a header describing how to decode the remaining contents of the file, followed by a number of segments containing the actual data. Some of the contents of this chapter will, by necessity, bear some resemblance to the documentation of the bytecode format that comes with Parrot [PBC], though what is presented here is in many cases more in-depth – the Parrot documentation is often quite sparse.

3.1 Header

The PackFile header contains the parameters that went into creating the file when the bytecode was stored, in addition to identification markers so that the program loading the bytecode can verify that the file is indeed a Parrot PackFile, and that it has not been corrupted.

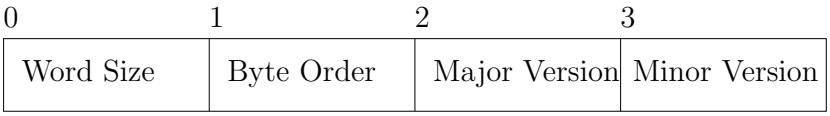

Figure 3.1: Bytecode Header, Bytes 0 - 3

Figure 3.1 shows the first four bytes in a PackFile. The first byte describes the word size used when dumping the bytecode to this file, in the unit of number of bytes in a word. At the time of writing, Parrot supports 32-bit and 64-bit word sizes, making values of 4 and 8 permitted for this field. The second byte tells the bytecode loader the endian type to use for word loading (the header is byte-orientated to escape endianness-issues). Permitted values are 0 for little-endian and 1 for big-endian. Bytes three and four contain the major and minor version of Parrot used for generating the PackFile (this is necessary, as there are usually incompatible changes between versions).

The next two bytes (five and six), as seen in figure 3.2, contain the size of inline integer constants and the floating point type. The integer size will be 4 or 8 bytes, usually matching the specified word size. The floating point type can either be 0, meaning IEEE 754 8 byte double, or 1, which is i386 little endian 12 byte double. In both these cases, and also for word size and byte order,

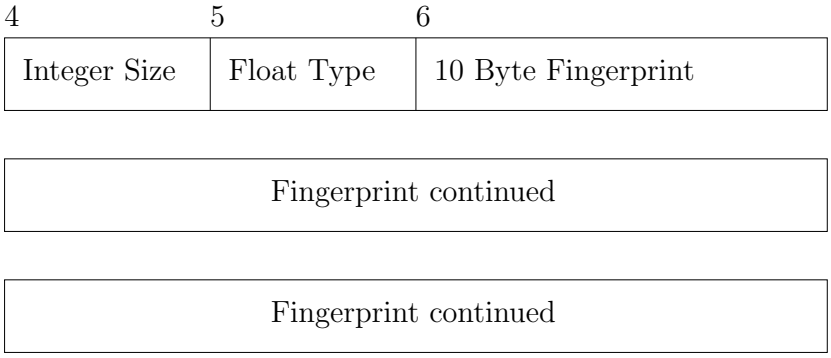

Figure 3.2: Bytecode Header, Bytes 4 - 15

the bytecode loader is expected to convert the various types to the appropriate platform native construct. The following ten bytes is a fingerprint of the core instructions supported by the version of Parrot used to create this PackFile.

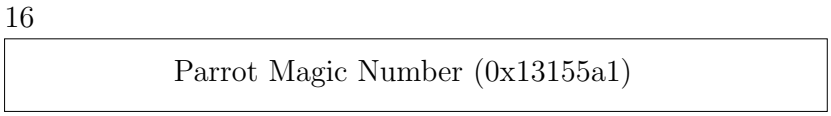

Figure 3.3: Bytecode Header, Parrot Magic Number

Bytes 16 to 19 (shown in figure 3.3) contain the Parrot magic number (0x13155a1). If the bytecode loader comes across anything else in this slot, the file should be rejected as an invalid PackFile. The number must (if necessary) be converted to the native endianness of the platform before checking.

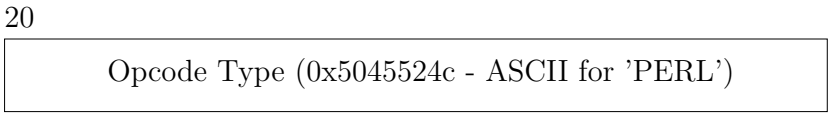

Figure 3.4: Bytecode Header, Opcode Type

The PackFile header ends with a word (with the size and byte order as specified above) detailing the opcode type. It is envisioned that in the future, Parrot might support multiple opcode formats, but at the time of writing, only one opcode type is understood. Parrot opcodes are signified by this field containing the number 0x5045424c, ASCII for "PERL". This is shown in figure 3.4.

3.2 Bytecode Format 1

The remainder of this chapter will deal with *PBC Format 1* (which replaced format 0 some years ago). Figure 3.5 shows the Format 1 header. Note that the byte count now depends on the word size. From here on, whenever byte numbers are used in figures, it is assumed that the bytecode was created on a 32-bit machine.

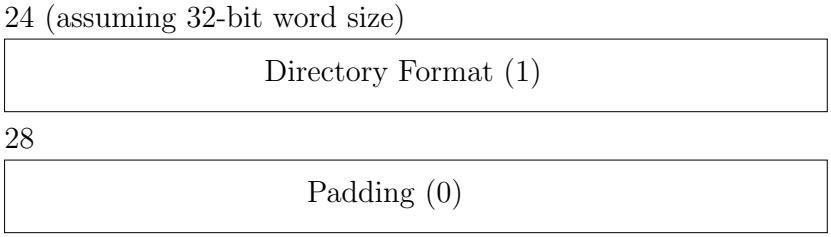

Figure 3.5: Format 1 Header

The padding shown in figure 3.5 is there to ensure the next segment starts on a 16-byte boundary. All Format 1 segments are so aligned.

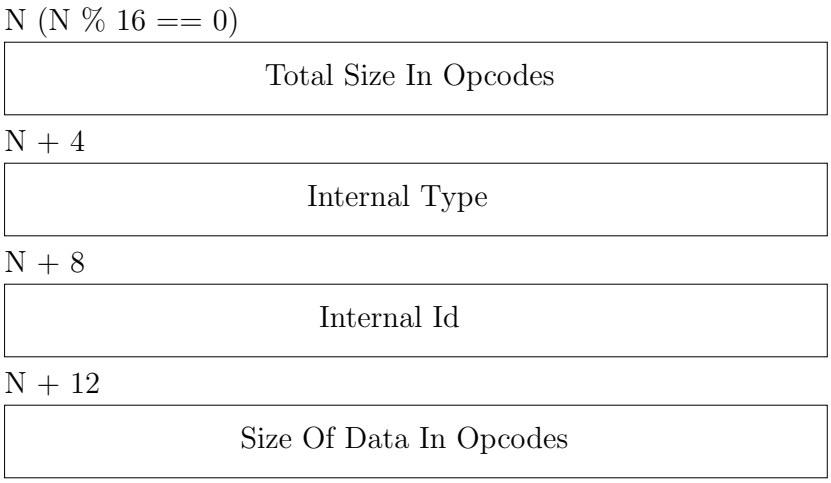

Figure 3.6: Format 1 Segment Header

All segments have a common header, as can be seen in figure 3.6. This header contains both the total size of the segment (in opcodes – that is, words) and the size of the payload following the header (which may be 0). There are two reasons for having both of these sizes. First, segments are 16-byte aligned, so there must

Type	*Description*
0	Directory Segment
1	Unknown Segment
2	Fixup Segment
3	Constant Table Segment
4	Bytecode Segment
5	Debug Segment

Table 3.1: PackFile Segment Types

be at least four words in the header (in case there is no payload). Of course, the last word could have been just padding, so the second reason is extra safety checks. The bytecode loader should ensure that $totalsize == payloadsize + headersize$. The *type* and *id* fields are used internally by Parrot, and are both zeroed in the PackFile.

After this header comes the optional payload, depending on the type of segment. Table 3.1 lists the available types. Each of these will now be discussed in more detail.

3.2.1 Directory Segment

Directory segments, unsurprisingly, act as a directory of segments. That is to say, it contains a list of what other segments are in the PackFile, what kind of segment each is, and how to find them.

Figure 3.7: Directory Segment, Number Of Entries

Since the directory segment is a simple list, the first word of the payload (figure 3.7) contains the number of entries. Following this are the entries themselves (figure 3.8). These consist of four pieces of information:

- the segment type. Again, see table 3.1 for the various types,
- the name of the segment. This is a nul-terminated string ("C string"), zero-padded to the nearest word size alignment,

Type	*Description*
'n'	Number Constant
's'	String Constant
'k'	Key Constant
'p'	PMC Constant

Table 3.2: PackFile Constant Types

- the offset in bytes where this segment can be found in the PackFile,
- and finally, the size of the segment (this is used for integrity checks – it must match the value given in that segments Format 1 header).

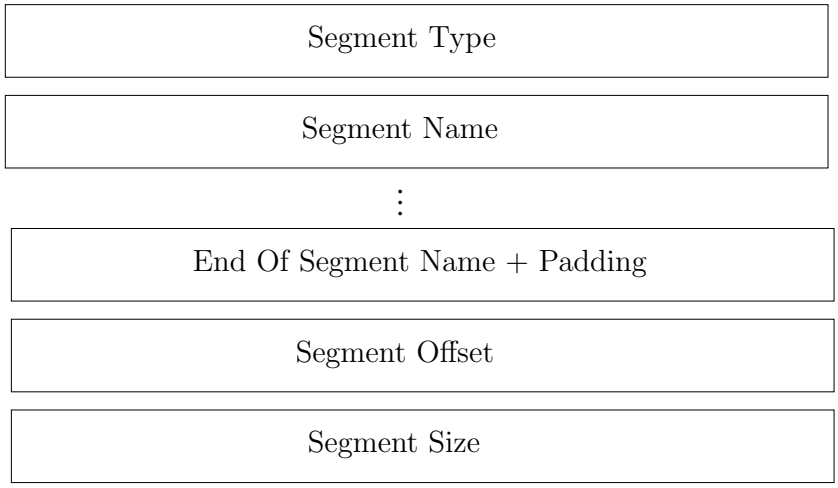

Figure 3.8: Directory Segment, Directory Entry

3.2.2 Constant Table Segment

Parrot bytecode deals with five types of constant. Of these, integer constants are placed inline in the bytecode stream, while the others go in the constant table segment. These four types are listed in table 3.2.

As with directory segments, a constant segment starts by stating the number of constants (see figure 3.9), and goes on to list the constants themselves, as in figure 3.10.

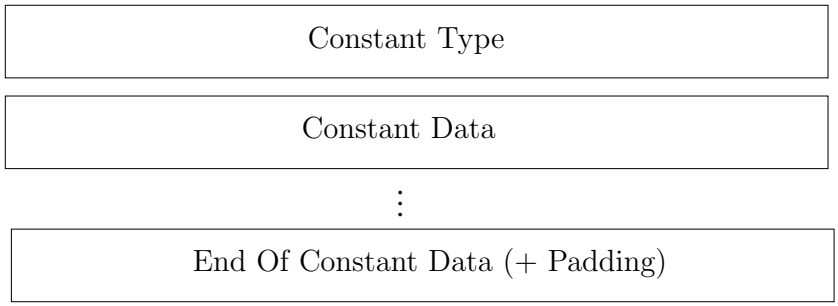

Figure 3.9: Constant Segment, Number Of Constants

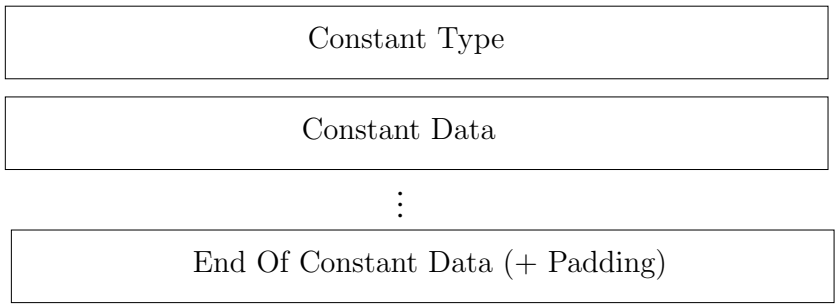

Figure 3.10: Constant Segment, Constant

Each constant entry containes some amount of data, depending on the type of constant. Number constants are the simplest ones, containing either an IEEE 754 8-byte double, or a 12-byte i386 little-endian double, depending on the platform the bytecode was generated on (see section 3.1). This is shown in figure 3.11.

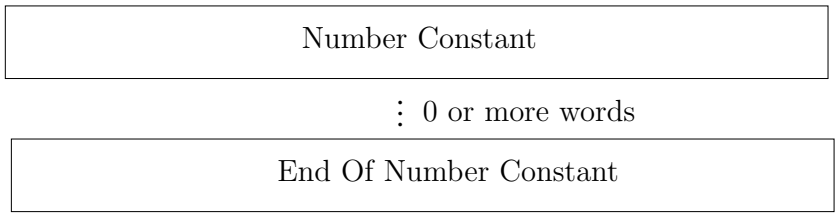

Figure 3.11: Constant Segment, Number Constant

String constants are somewhat more complex. Figure 3.12 shows how a string constant is laid out in the PackFile. The *Parrot string* structure contains information on the character set of the string, any special flags, and of course the string data itself. Note that Parrot constant strings are not nul-terminated, and can therefore contain the nul character.

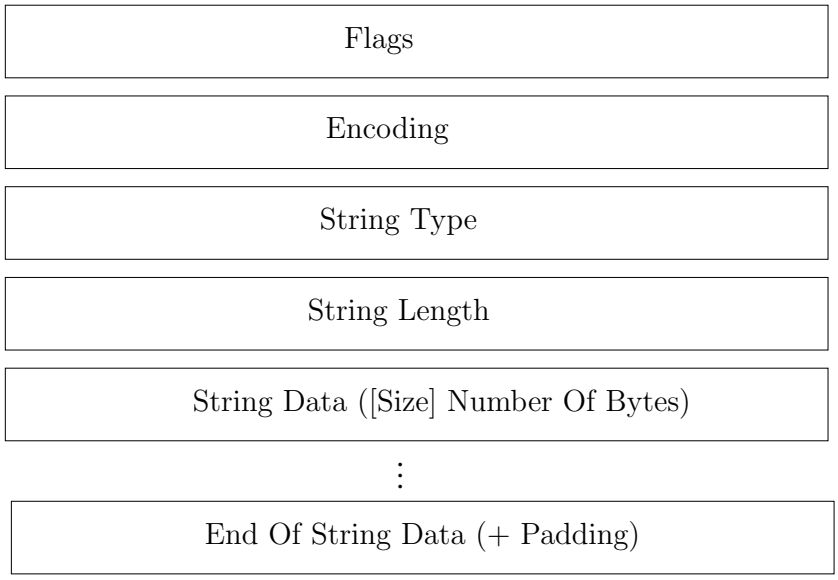

Figure 3.12: Constant Segment, String Constant

Key constants are used in many PIR instructions, in places where a PMC is accessed as an array, hash or similar. See figure 3.13 for an example of how key constants look like in PIR.

```
P0 = new .Hash
P0["foo"] = "bar"
```

Figure 3.13: Key Constant Example

```
P0["foo";"bar"] = "baz"
```

Figure 3.14: Multi-component Key Constant Example

In the bytecode, key constants appear as in figure 3.15. A key can be composed of several components (the example given coul dfor instance be changed as in figure 3.14 – though this would be an invalid index for the Hash PMC). The possible component types are listed in table 3.3 (integer constants are, as mentioned above, inline in the bytecode stream, and uses a special "integer key constant" type rather than being listed in the constant table).

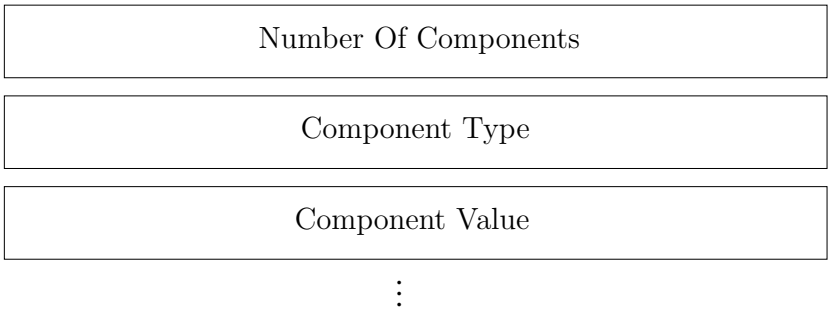

Figure 3.15: Constant Segment, Key Constant

Type
Integer Register
Number Register
Number Constant
String Register
String Constant
PMC Register

Table 3.3: Key Constant Component Types

The value part of the key component is either a register number or a pointer into the constant table.

Finally, PMC constants contain frozen (serialised in Java-lingo) PMCs. There is a poorly-documented common header, but as parts of it remains a mystery, even after long sessions of digging around in the Parrot source code, it is not included here. Most of the work that needs to be done to read a PMC constant is done by the various PMCs themselves in their thaw() method.

3.2.3 Bytecode Segment

Bytecode segments carry only actual executable bytecode as their payload. The possible pieces found in the bytecode segment are shown in figure 3.16. Subroutine PMC constants index into the bytecode segment, telling the interpreter where to start executing.

3.2.4 Debug Segment

A debug segment contains mappings from source code line numbers to offsets into the bytecode. There are two types of mapping: 0, which means there is no source

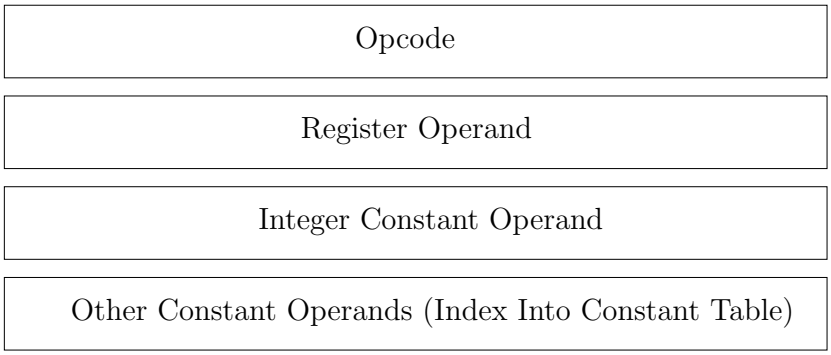

Figure 3.16: Bytecode Segment Constituents

available for the bytecode at the given offset, and 1, which means that the source code is available in a file. For the latter case, an index into the constant table is provided to locate the filename containing the source code. Figure 3.17 shows the PackFile layout for the debug segment.

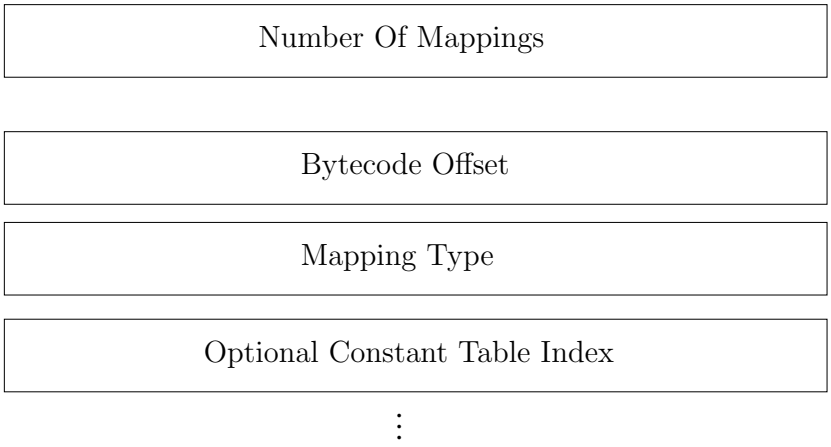

Figure 3.17: Debug Segment

3.2.5 Fixup Segment

Finally, there is the fixup segment. The main use for this segment is to list the subroutines present in the bytecode and map subroutine name to constant table index for these. This is fixup type 1 in figure 3.18. Fixup type 0 is used for labels that index directly to bytecode offsets.

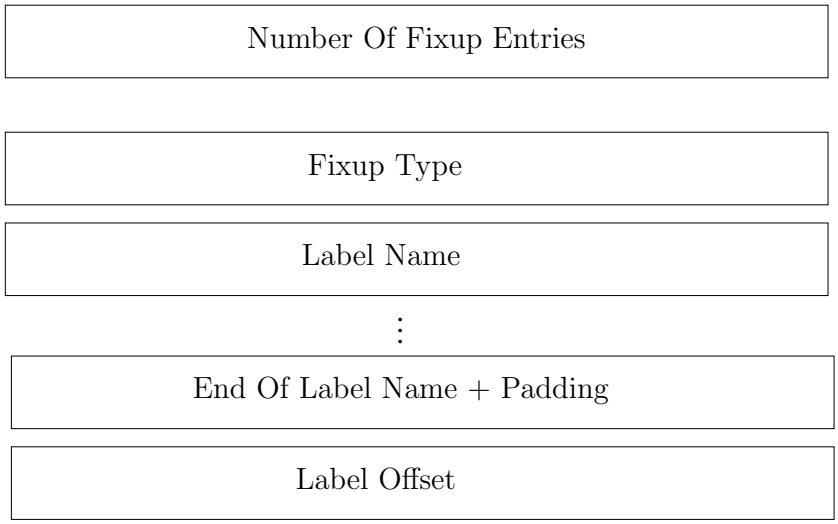

Figure 3.18: Fixup Segment

3.3 Summary

- Parrot bytecode files (called PackFiles) start with a header that contains things necessary for loading the remainder of the bytecode. Examples of these are word size, floating point format, endianness and so on.
- The PackFile is divided into segments, such as a directory segment, a constant table, the bytecode segment itself, etc.
- There are five types of constant: integers (written inline in the bytecode), numbers (floating point), strings, keys and PMCs.
- The bytecode segment contains the directly interpretable bytecode stream.

Chapter 4

Parakeet

"Customer: 'E's not pinin'! 'E's passed on! This parrot is no more! He has ceased to be! 'E's expired and gone to meet 'is maker!

'E's a stiff! Bereft of life, 'e rests in peace! If you hadn't nailed 'im to the perch 'e'd be pushing up the daisies!

'Is metabolic processes are now 'istory! 'E's off the twig!

'E's kicked the bucket, 'e's shuffled off 'is mortal coil, run down the curtain and joined the bleedin' choir invisibile!!

THIS IS AN EX-PARROT!!

(pause)

Owner: Well, I'd better replace it, then."

Monty Python, the "Dead Parrot" sketch.

4.1 Overview

Of course, where this project is concerned, Parrot certainly is not dead, nor is it being replaced exactly (and hopefully, any comparison with the owners suggested replacement – a slug – can be avoided).

As stated in the introduction, the goal of the project is to extend a Java virtual machine to run Parrot bytecode. Specifically, the Jikes RVM has been targeted, for several reasons – some of which are:

- it is the preferred development JVM for the Jamaica project, the group offering the project,
- it is written in Java, and
- it has been extended in a similar fashion earlier, so lessions learned, and even some code, can be reused.

Having a second implementation of the Parrot virtual machine will hopefully prove benificial to the Parrot community. Once Parakeet gets up to speed (both featurewise and in actual raw performance) with Parrot, the competing projects

can egg eachother on, furthering both in ways that would not have happened with just a single implementation.[1]

The project should also, again, hopefully, prove to be of some value to the Jikes RVM. For instance, Parakeet provides a new user of various bits of the internal API, and will exercise those parts differently than their normal use. This may (though it has not yet) help discover bugs that would otherwise be harder to find. Also, Parakeet goes some way towards making the Jikes RVM a little less Java-centric – it is a *research* virtual machine after all, not just a *Java* virtual machine.

4.2 Jikes RVM

When the project was first started, the choice between generating Java bytecode or producing Jikes RVM HIR directly had to be made. Two things made this choice easier:

- first, the register-based architecture of Parrot maps fairly well onto Jikes RVM HIR, whereas it would be more of a pain if stack-based Java bytecode had to be generated. In fact, many parts of Parrot bytecode might seem to match the Jikes RVM internals better than Java bytecode even, and
- second, the Jikes RVM has a substantial API for generating HIR. If, in stead, Parakeet was to translate Parrot bytecode into the Java equivalent, most everything would probably have to be done by hand.

Sadly, not all features of Parrot are as well fitted for the Jikes RVM. Instructions like setting the value of registers indirectly – by register number, or branching to a bytecode offset stored in a register, are features necessary for dynamic languages like Perl 6 or Python. Many such (and similar) features are not catered for at all in Java, and hence require a bit of extra work to fit them in.

[1] That is the authors pipe dream anyways...

4.3 PearColator

It should be noted here that quite a bit of the underlying code that does the tie-in with the Jikes RVM is reused or adapted from the dynamic binary translator PearColator [Mat04]. This includes the "fake method" technique for compilation as well as the various patches required for hijacking and replacing internal bits of the JVM.

Having access to the PearColator source was also very helpful in the first days of the project, as it provides examples of how various parts of HIR are generated. This in a more contained setting than the bytecode to HIR converter in the Jikes RVM source tree.

4.4 Parakeet Architecture

A very brief overview of the architecture used in Parakeet is shown in figure 4.1. This section will now go on to describe some of these in more detail.

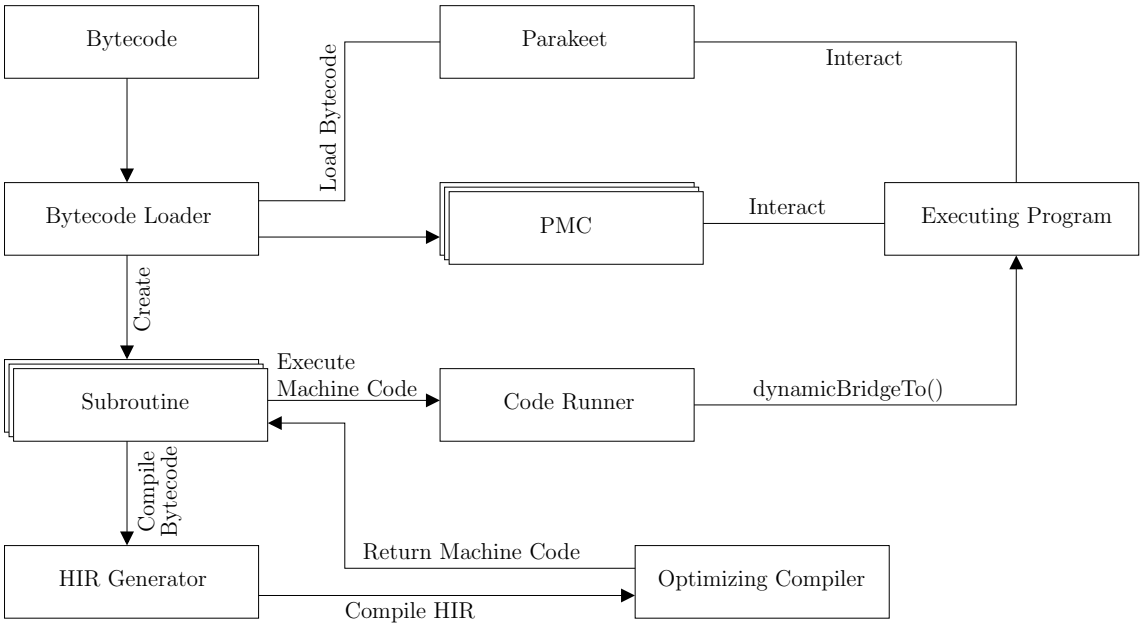

Figure 4.1: Parakeet Architecture Overview

4.4.1 The Bytecode Loader

The first part of the architecture invoked when Parakeet is run is the bytecode loader. This parses the PackFile format, as specified in chapter 3, and performs various sanity-checks.

When necessary, the bytecode loader interacts with a multitude of different Parrot Magic Cookies, and with Parakeet itself. Global constants are registered, namespaces are setup and subroutines within those namespaces are discovered and stored in such a way as to be easily reachable later.

Much of the work of the bytecode loader is delegated to the SegmentData class, which in turn is a thin wrapper around Javas ByteBuffer. Once the SegmentData structure is properly setup, it is easy to read complex types like Parrot strings from the PackFile. SegmentData is also responsible for handling endianness issues, word sizes and the floating point types used, converting, when necessary, to Java types.

4.4.2 Subroutines

Once the bytecode has been loaded, Parakeet will scan through all the found subroutines looking for the one with the `:main` flag set. If none of the subroutines have that flag, the one listed first in the bytecode will be assumed to be the main subroutine. Then that subroutine is invoked.

When a subroutine is invoked, it instantiates a subclass of VM_NormalMethod (this subclass is, confusingly perhaps, also called Subroutine) designed to hijack a method in a dummy class. The Subroutine class supplies its own HIR generator in stead of the default Java bytecode one. The Subroutine is then compiled, and the resulting code is handed over to the code runner.

4.4.3 HIR Generator

The HIR generator is really the core of Parakeet. This is where the action is at. The HIR generator is what actually reads the bytecode stream and uses the Jikes RVM API to create code that can be compiled and run.

Aiding in this task are a set of opcode decoders. HIR generation is run in a loop which calls out to the decoders until there is nothing left of the bytecode to process. The opcode decoders in turn call back into the HIR generator to actually perform the work. The decoders are designed to be capable of dynamically loading, at runtime, mimicing Parrots loadable opcode libraries.

4.4.4 The Code Runner

The code runner is little beyond a method with the correct return type and parameters required by Parrot subroutines. This method simply calls VM_Magic.dynamicBridgeTo which changes the JVMs frame of execution into that of the code supplied by the HIR generation / compilation phase.

4.4.5 Parrot Magic Cookies

PMCs are used several places in Parakeet. Not only are they available for programs to use, they are used heavily internally. Arguments to subroutines and return values are both PMCs, constants in the bytecode that tell how to interpret those arguments and return values are PMCs, even subroutines themselves are PMCs.

The magic cookies are used both at HIR generation time and at runtime. Everything from namespaces to global variables, lexical structures to stacks are represented or emulated by different types of PMC. In Parrot, even the interpreter is a PMC, though this is not used by Parakeet. PMCs are, like the opcode decoders, dynamically loadable.

4.5 Type and Method Resolving

The Jikes RVM uses the original bytecode array from the dummy method (that gets hijacked by the compilation phase) for resolving purposes. Anything that is not in the boot image of the JVM may potentially have to be resolved at runtime, and the Jikes RVM uses the bytecode to discover the correct types to load.

This leads to quite a bit of "cruft" in this dummy method. Every method call that can be generated by the HIR generator must have a corresponding

bytecode index, in case it needs to be dynamically resolved. The ResolveHelper class contains mappings from method signatures to bytecode indices. This class is automatically generated by a script that inspects the dummy method, adding an extra compilation step whenever methods that need to be resolved are added.

In the future, Parakeet should probably provide a better way of doing such resolving, changing the Jikes RVM not to use the Java bytecode associated with the method that is being compiled. Of course, once foot is set on that path, there are probably many things that should be done for better integration. See chapter 6 for more on future work.

4.6 Progress

At the time of writing, Parakeet is not yet a suitable replacement for Parrot. Indeed, there is quite a way yet to tread. There are several reasons for this, some of which are:

- the scope of the Parrot project was not entirely understood when setting out, and proved to be quite a bit larger than first believed,
- that scope has been changing quite a bit underways, and
- even the parts that have remained stable tend to have very variable amounts of documentation, and much of the documentation itself is out of date, making for a lot of tedious reading of the Parrot source.

More than half of Parrots 1200-odd opcodes have been implemented so far, starting with the basic, core ones require to get even the most minuscule test programs running, and expanding outwards whereever seemed prudent (or most interesting, it must be admitted) at the time. In addition, about a fifth of the Parrot core PMCs are (to varying degrees) implemented in Parakeet. Most of these were implemented because they are used internally by Parrot (frozen in the bytecode for instance) at some point or other.

4.6.1 Playing catch up with Parrot

One of the more frustrating aspects about the project has been the ever-changing scope of Parrot. Each new release has brought new, different, incompatibilities with older versions (and thus with Parakeet). Whenever a new release would come out, everything else would have to be put aside for some time to make Parakeet handle things the same way as Parrot.

In fact, the Parrot developers have gone as far as saying they care nothing for backward compability at all on the bytecode level. Compability is kept at PIR code level, which, with PIR being what compilers that target Parrot should generate, admittedly does make some sense. This way, the Parrot team can fix problems in the lower levels without affecting developers that do not care about Parrot internals (which probably would be most of them). A recent change of leadership in the project has also done its share to turn things upside down.

As an example of changes made, the 0.4.5 release of Parrot added about 50 opcodes, interleaving the new opcode numbers in-between old ones to make it utterly incompatible. This release also changed what types of PMCs were used for various things frozen in the bytecode. Incidentally, this is the version of Parrot that Parakeet is currently targeting – 0.4.6 (the first release after the above-mentioned leadership change) has been out for some time now, but there has not been time to incorporate the massive changes to core features, like namespaces, into Parakeet yet.

4.7 Testing

Testing of the project has been performed through the use of a home-grown unit testing system. As of writing, more than 280 tests exist in the source tree, all of which currently pass.

The tests written so far are not very extensive though. Mostly, they test only whether a feature works at all or fails utterly, and will often fail to exercise the many corner cases. Also, although each test is only a tiny PIR program, the way they are run is quite inefficient – a separate instance of the virtual machine is run for each test, which makes for a bit of a wait when running the test suite.

Improving on the test suite and tools will be important for the future well-being of the project.

It is hoped that, when the Parakeet implementation has matured somewhat, something like the xUnit framework can be implemented in pure PIR. This will make testing much easier than it is today, while at the same time running more efficiently. Hopefully, this will result in more tests being written, thus exposing more bugs and increasing trust in the code base.

4.8 Obtaining Parakeet

A SourceForge project has been set up for Parakeet. It can be found at **http://sourceforge.net/projects/parakeet/**. Currently, the Subversion repository at SourceForge is the only way of accessing the Parakeet source code. It should be warned that, at the time of writing, Parakeet should be considered alpha quality at best.

4.9 Summary

- Parakeet is the object of this project, an extension to the Jikes RVM to run Parrot bytecode.
- The project should hopefully prove to be of some benefit to all involved parties.
- A lot of Parrot maps well onto the Jikes RVM, though some of the more dynamic features has required more work to implement.
- Much of the Jikes RVM tie-in code was adapted from the PearColator dynamic binary translator.
- The Parakeet architecture attempts to span everything that is currently possible in Parrot, including dynamically loadable opcode libraries and magic cookies.
- Dynamic resolution has proven tedious, and work should probably be done in this area at some point.

- Parrot has been, and continues to be, a moving target.
- Parakeet is tested using a home made unit testing framework. Currently there are more than 280 tests in the source tree.

Chapter 5

Performance Testing

> ***"Why your own virtual machine? Why not compile to JVM/.NET?***
>
> *Those VMs are designed for statically typed languages. That's fine, since Java, C#, and lots of other languages are statically typed. Perl isn't. For a variety of reasons, it means that Perl would run more slowly there than on an interpreter geared towards dynamic languages."*
>
> **Question from the Parrot FAQ.**

5.1 Overview

As mentioned in the introduction, one of the goals for this project is to achieve good performance by reusing the optimizing compiler of a Java virtual machine. This chapter describes some attempts to benchmark Parakeet, compares its performance against Parrot and tries, to some extend, to explain the results.

The benchmarks used come from the Parrot distribution. Quite a few more benchmarks are shipped in the `examples/benchmarks/` directory of the Parrot source tree, but many of these depend on features not yet implemented in Parakeet (that is, unimplemented opcodes in most cases). Of course, even if all the benchmarking programs shipped with Parrot could have been run, they do not form a comprehensive benchmarking suite like e.g. Dhrystone. Regrettably, this means that a very limited set of features will be performance tested herein, so while the tests do give some pointers as to how Parakeet performs compared to Parrot, the conclusions drawn from the data should be taken with the proverbial pinch of salt.

Finally, it should be mentioned that Parakeet development so far has been a great deal more geared towards actually getting Parrot features implemented than worrying about how well those features actually perform. Shortcuts that might lead to loss of performance have been taken where they could save time on actually getting something working (under the principle of *"Make it work, make it right, make it fast"*).

5.2 Benchmarking Environment

All data collected is from an IBM T42 laptop with an Intel Pentium M 1.7GHz CPU and 1GiB of memory. The tests were run on Linux 2.6.17 with a (constant) minimal set of other applications loaded at the same time.

All benchmarks were run at least 20 times for each input value, often more, and the results were then averaged. This to ensure that the program (and virtual machine) were hot in cache, and to even out jitter caused by other processes running at the same time.

5.3 Fibonacci Benchmark

This benchmark calculates the Nth Fibonacci number. Because a recursive algorithm is used, method call (or subroutine call for Parrot) overhead is likely the most prominent factor for performance.

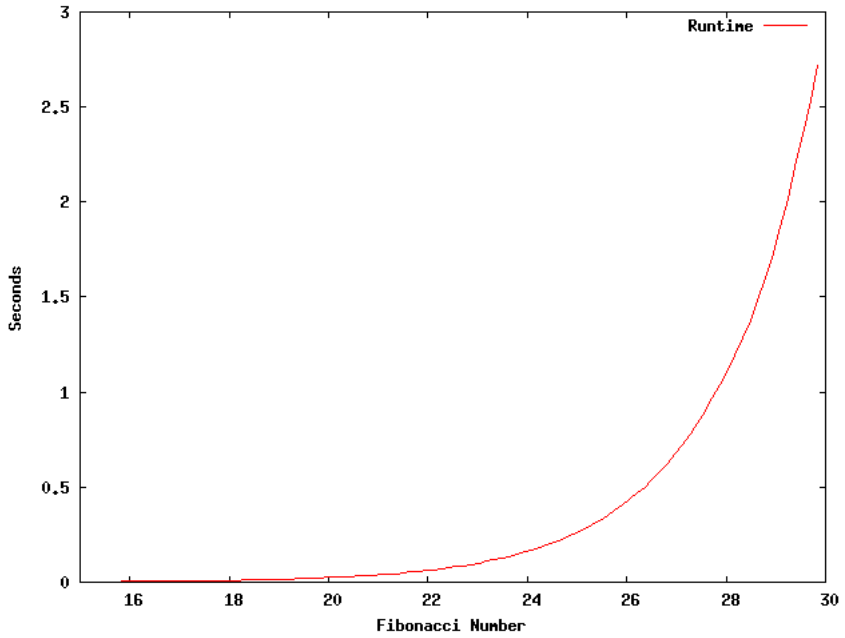

Figure 5.1: Parrot Fibonacci Number Performance

Figures 5.1 and 5.2 show the runtime for the Fibonacci program generating the 15th to 30th Fibonacci number, running on Parrot and Parakeet respectively. As

can be seen, both plots more or less have the same shape (owning to the $O(N^2)$ algorithm used in the program). This leads to the point that there is apparently no algorithmic difference in how Parrot and Parakeet execute this program. When Parakeet proves less efficient, it is because each (or at least one) step runs slower.

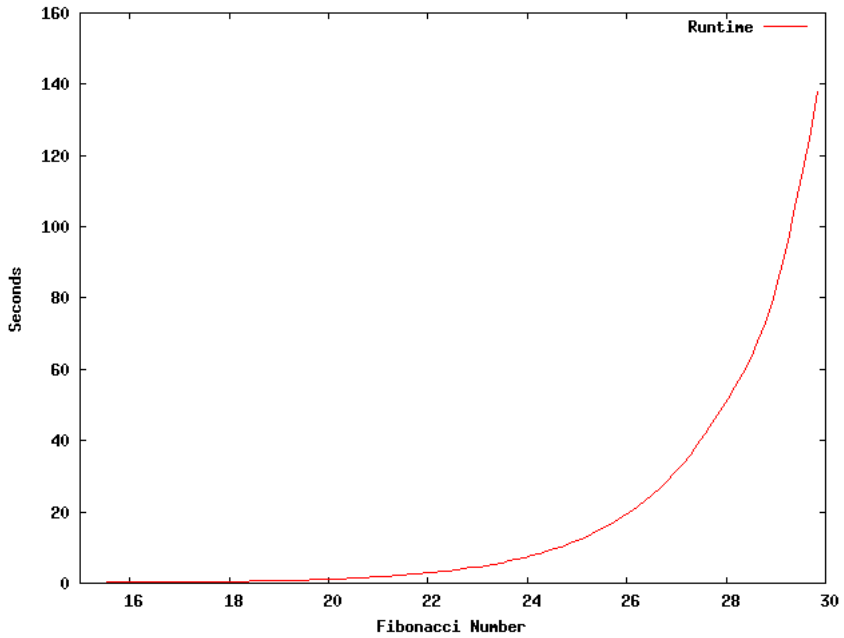

Figure 5.2: Parakeet Fibonacci Number Performance

Figure 5.3 illustrates how much slower Parakeet runs the Fibonacci program. For the first 15-20 Fibonacci numbers, the load time of the virtual machine plays a significant part (and Parrot obviously has the shorter startup time). After this though, the difference in performance is more or less evened out, with Parakeet using about 45 times longer to execute the same program.

Quite a bit of this rather large gap in performace can likely be attributed to the fact that subroutine calls in Parakeet have much greater overhead than Parrot. Obviously, there are great gains to be had from work in this area.

5.4 Prime Number Calculation

The next benchmark run was one that checks all the numbers up to a set limit for primeness. The program uses Integer PMCs, so a great deal of the runtime

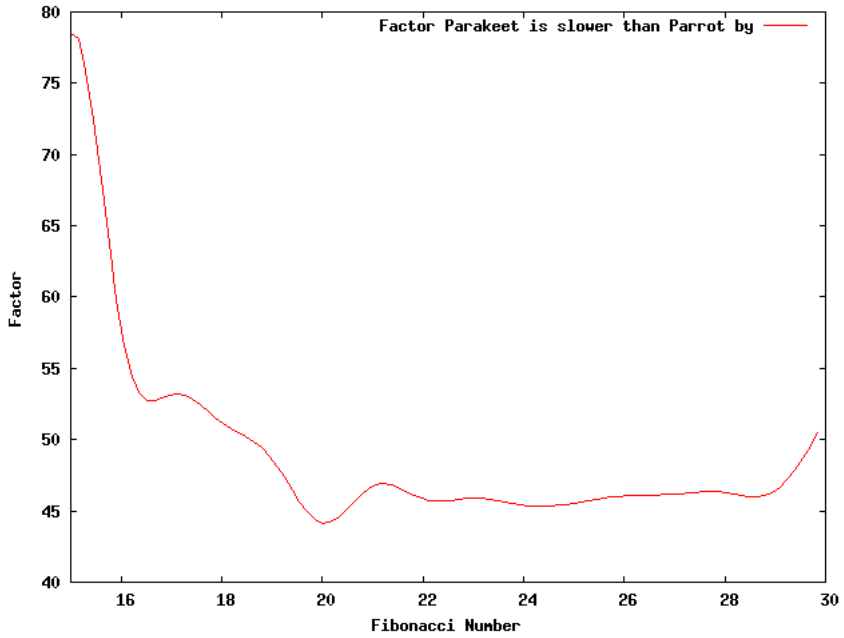

Figure 5.3: Relationship between Parrot and Parakeet Fibonacci Performance

will be spent on mathematical PMC operations (boiling down to method calls and integer arithmetic in Parakeet).

As can be seen from figures 5.4 and 5.5, Parrot again beats Parakeet hands down. Like the Fibonacci benchmark, an $O(N^2)$ algorithm is involved, and again the plots are quite similar. Parakeet simply uses more time to execute each step.

Figure 5.6 shows that Parakeet takes on the order of about 20 times as long to run the primeness checks. Unlike the Fibonacci benchmark though, this time there are no subroutine calls. The main culprit is likely to be the PMC methods – the particular set of operations used by the program causes several method calls, object instantiation and integer arithmetic for every iteration. Especially the object instantiations appear to make performance take a big hit.

Like mentioned before, the easiest path to get features going has been chosen a number of times during development. Mathematical PMC operations could certainly be implemented differently, and should apparently be reimplemented for efficiency reasons some time in the future.

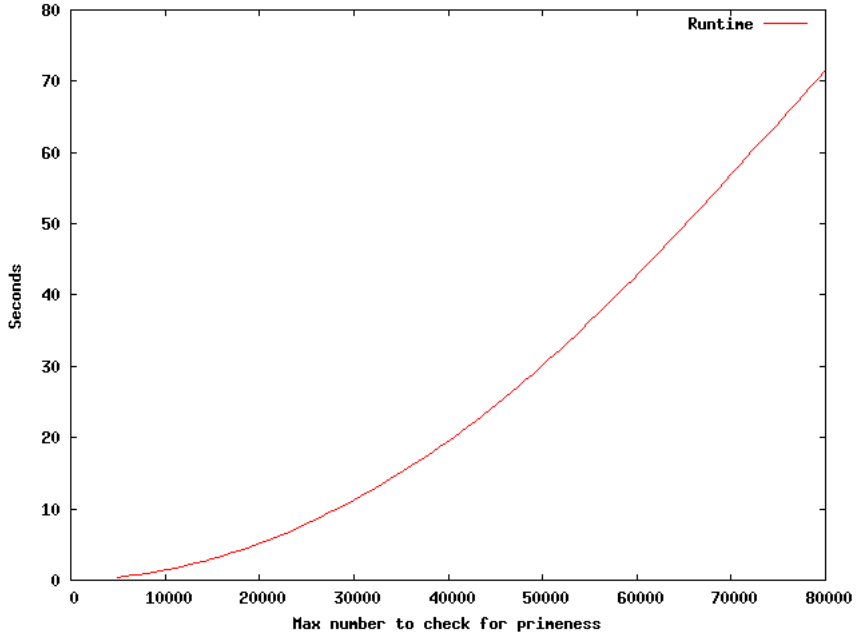

Figure 5.4: Parrot Prime Number Calculation Performance

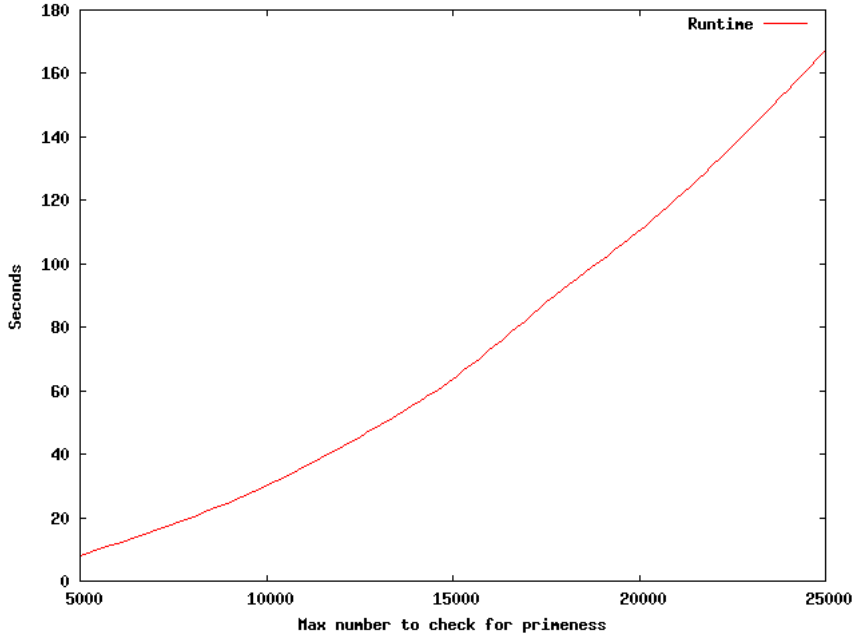

Figure 5.5: Parakeet Prime Number Calculation Performance

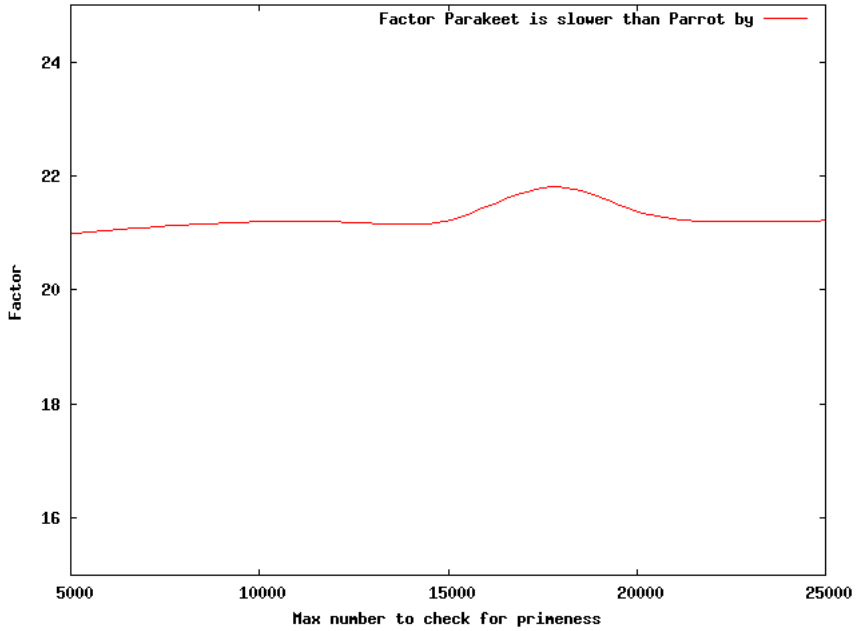

Figure 5.6: Relationship between Parrot and Parakeet Prime Number Performance

5.5 Variable Argument Subroutines

The variable argument subroutines benchmark is quite simple. It consists of a subroutine that is called N times from a loop. The subroutine arithmetically adds all its input parameters and returns the result. The parameters passed in are PMCs of types Integer, Float and String.

Again Parakeet loses out to Parrot, as can be seen in figure 5.7. Also, again, the factor of how much slower Parakeet is, is constant when the number of iterations is high enough to ignore startup time. Like with the prime number calculation, that factor is about 20. This is shown in figure 5.8.

5.6 MOPS

Finally, when all seems lost on the performance front for Parakeet, enter the simplest benchmark of them all: set a variable to some large number, then decrement and iterate until it reaches zero. This makes for a crude measure of MOPS (Million Operations Per Second). There are two versions of this in the Parrot distribution, one using integer registers and one using Integer PMCs. As will be

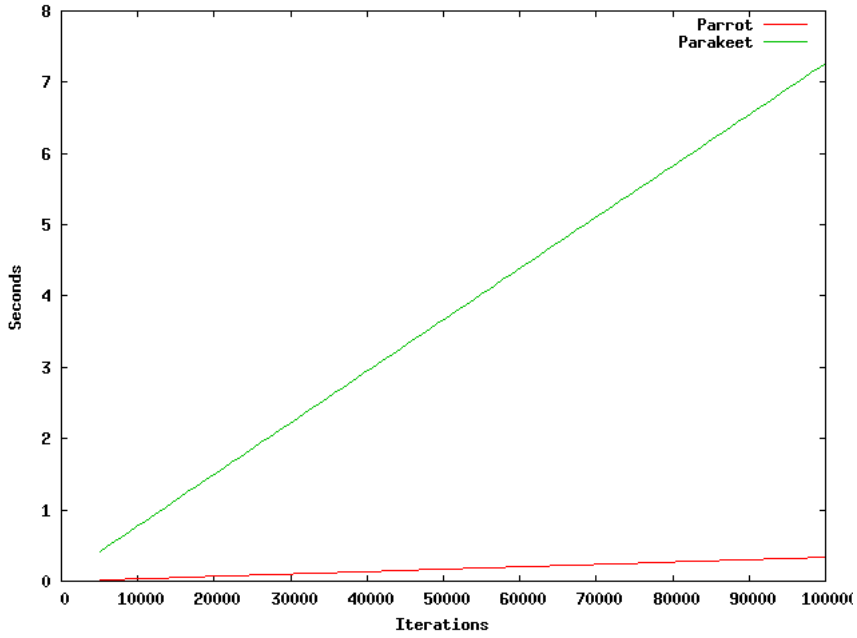

Figure 5.7: Variable Argument Subroutines Performance

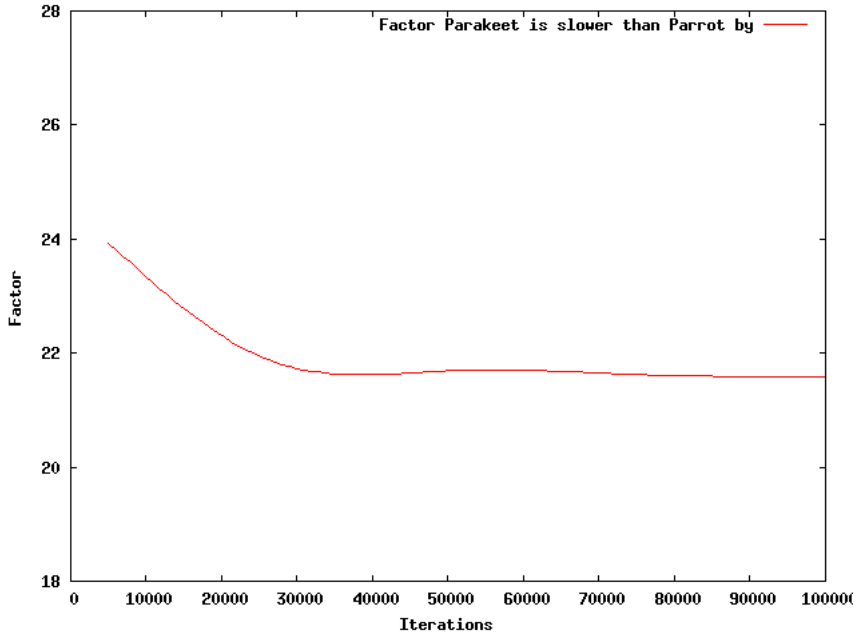

Figure 5.8: Relationship between Parrot and Parakeet Vararg Subroutines Performance

seen, the results vary wildly.

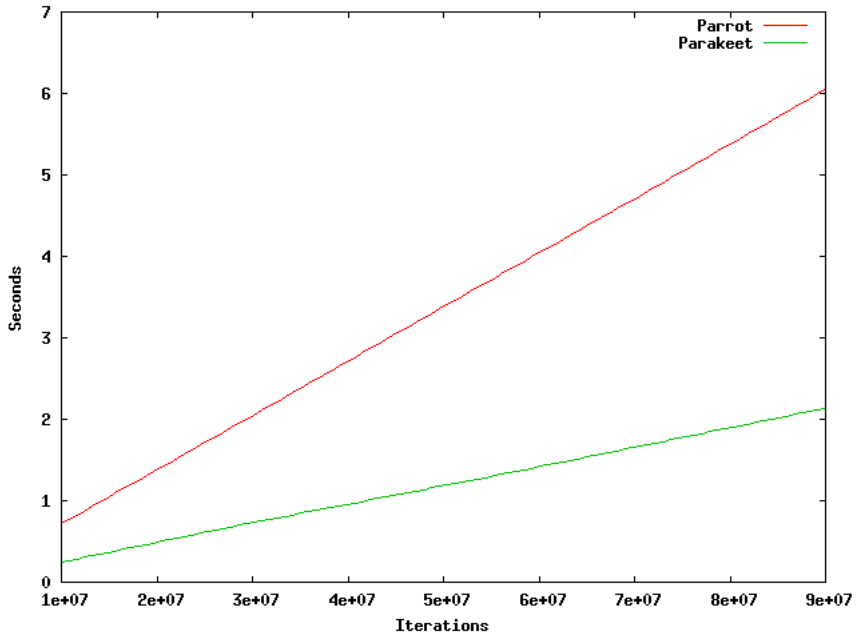

Figure 5.9: PMC MOPS Performance

Look first at the PMC performance in figure 5.9. Finally, a glimmer of hope for Parakeet! Unfortunatly, it does not hold for the integer register version of the benchmark. See figures 5.10 and 5.11 – note the difference in scale on the Y axis!

For the PMC version, Parakeet outperforms Parrot by a factor of three. With integer registers, it is more than turned the other way around, with Parakeet taking as much as 95 times as long for the same task (see figure 5.12). The PMC-based program takes advantage of opcodes that permutate the PMC value in stead of replacing it, which, it seems, is taken advantage of in a huge way in some optimization phase in the Jikes RVM.

Finally, in figure 5.13, the MOPS for both tests, with both virtual machines, are compared.

5.7 Summary

- The benchmarks used come from the Parrot distribution. More benchmarks than those presented here are available, but unfortunately, most of them

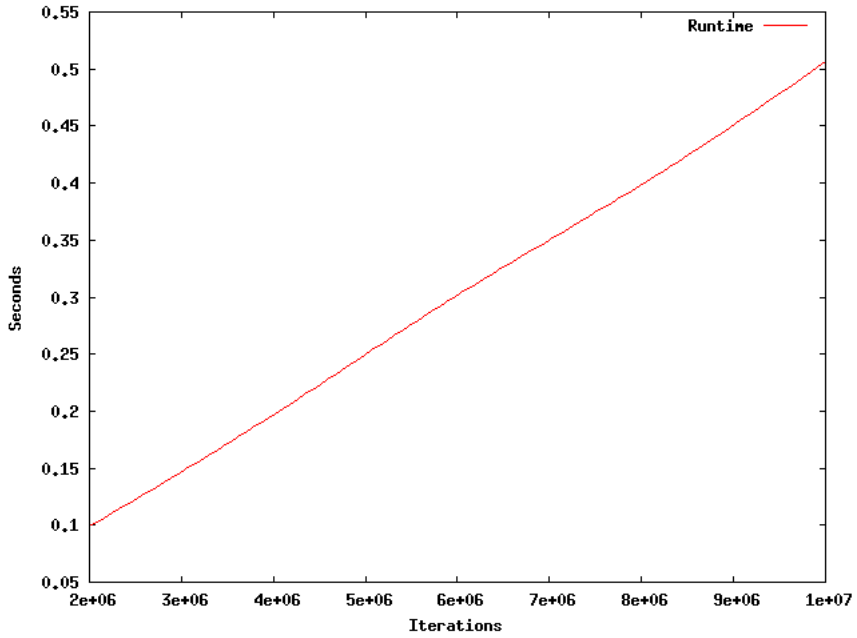

Figure 5.10: Parrot Integer MOPS Performance

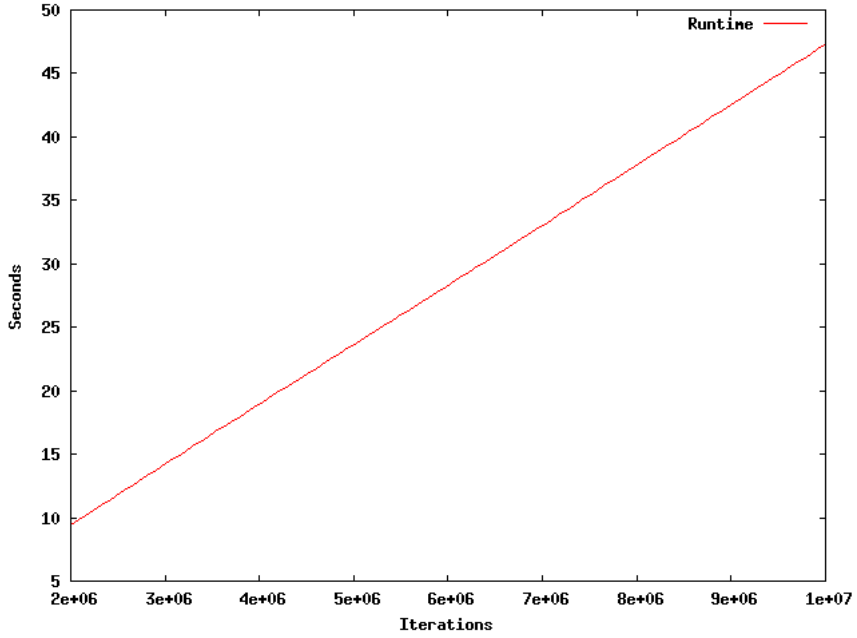

Figure 5.11: Parakeet Integer MOPS Performance

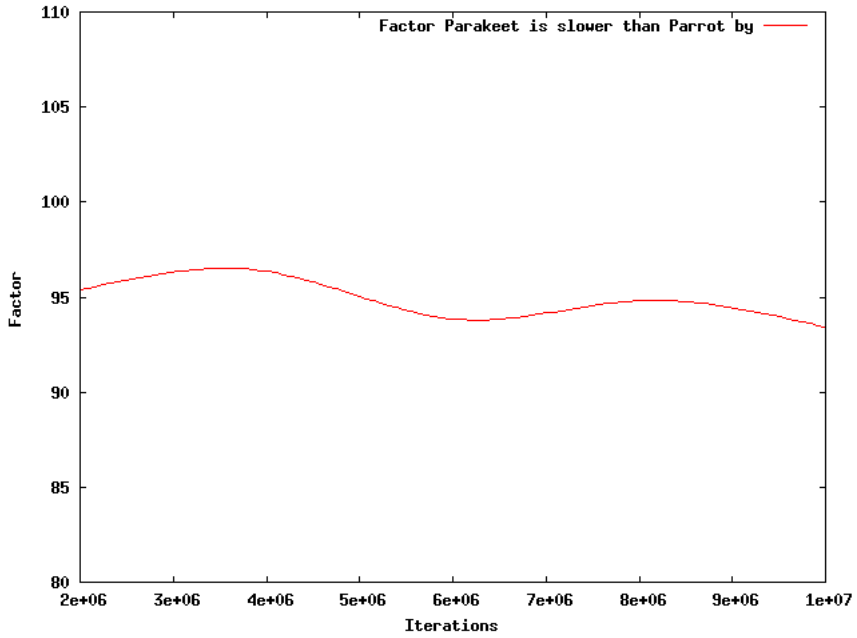

Figure 5.12: Relationship between Parrot and Parakeet Integer MOPS Performance

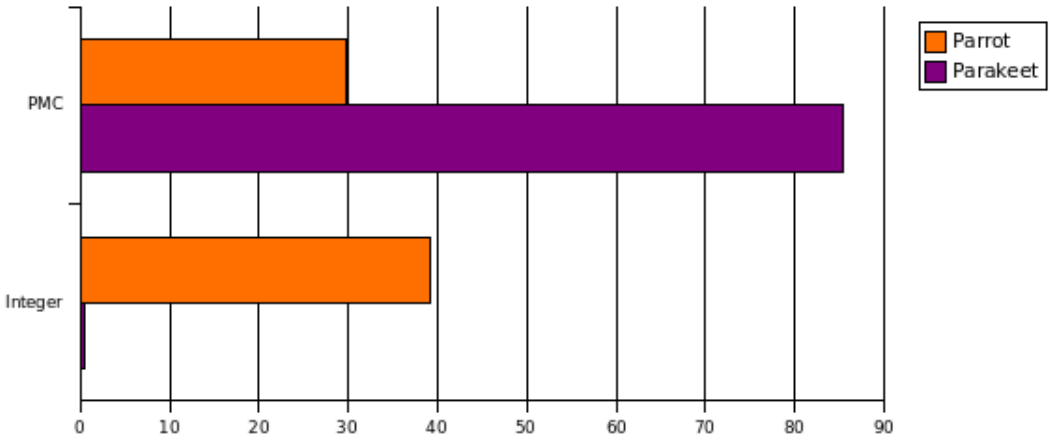

Figure 5.13: MOPS Performance Comparison

depend on features not implemented in Parakeet.

- Because of the limited number of programs available, only a limited feature-set is performance tested.
- Parakeet has not yet been optimized. Development so far has been geared towards making features work, not dwelling long enough on each to make them actually perform well.
- Every benchmark run except one has shown Parrot beating Parakeet utterly, the latter being between 20 and 100 times slower in most cases.
- The one performance test where Parakeet did beat Parrot, while gratifying, constitutes only shaky proof that Parakeet can actually compete in any way at this stage.

Chapter 6

Conclusion and Future Work

6.1 Overview

The Parakeet implementation of Parrot is coming along well so far, though there are many, many things left to be done for a Parakeet that can do everything Parrot can. Almost half of the opcodes remain unimplemented, and by far the majority of PMCs are untouched.

Also, the performance benchmarking done, although not very comprehensive, has shown that there is a long path ahead indeed if Parakeet is going to beat Parrot at its own game (remember, one of Parrots intentions is to outperform virtual machines designed for statically typed languages when running programs written in more dynamic languages). There are also areas of Parrot that have gone entirely unexplored so far, like for instance the various compilers, or native library interactions.

6.2 Completing the Opcode Library

A coverage of only slightly above 50% of the available opcodes might not seem very impressive. It should however be noted that many of the unimplemented instructions are either specific to certain languages (for instance, both Python and .NET have their own blocks of opcodes), specific to the internals of Parrot (several opcodes should actually never occur in a bytecode stream, but are only for internal use) or esoteric mathematical instructions.

There are, however, areas where work is sorely needed. Especially I/O instructions are heavily under-represented in the set of implemented opcodes. The upside is that many of these opcodes should prove easy to implement – only a lack of time has kept them back so far.

6.3 Parrot Magic Cookies

The PMCs also are in need of attention. The situation is, however, like with the opcodes, not quite as dark as it might seem. Many of the unimplemented PMCs are merely variants of those that are, like fixed (unresizable) arrays, or arrays that deal only with integers, and so on, and will, when implemented, inherit much from the existing PMCs, with only a minimal set of their own unique features.

Something else that needs to be dealth with when it comes to PMCs, though, is the inheritance hierarchy. At the moment, the situation is muddled, and requires some house cleaning. Preferably sooner rather than later, as the more PMCs are implemented, the more of a mess it will be to clean up later. Once that task is accomplished, much code duplication within the PMCs should vanish.

6.4 Performance

As was shown in chapter 5, in nearly all cases, Parakeets performance is quite dire. Really, for there to be much value (for end users at least), performance needs to be put in the front seat. As has been explained earlier, up to this point in development, it has been mainly a race to get features working, undoubtably leading to some corners being cut and damaging performance.

Indeed, this race is likely to go on for some time yet, until enough of Parrot has been implemented to make it worthwhile to concentrate on performance. This saying has been mentioned earlier, but bears repeating: *Make it work, make it right, make it fast!*

6.5 Parrots Compiler Suite

As was mentioned in the introduction, the early goal of the project was to run Perl 6 programs. This was thwarted by the fact that Perl 6 really is not a finished language, and the Parrot compiler is not even up to speed with the specification as it stands at the moment.

Nevertheless, and there are other compilers shipped with Parrot as well, it would be very useful to have the compiler suite running on Parakeet. This would help separate the project from its now heavy dependencies on Parrot. It would also finally enable end users to actually use Parakeet for something useful.

Bibliography

[AAB00] Bowen Alpern, C. R. Attanasio, and John J. Burton. The jalapeño virtual machine. *IBM Systems Journal*, 2000.

[chr05] chromatic. A plan for pugs. http://www.perl.com/pub/a/2005/03/03/pugs_interview March 2005.

[Coz01] Simon Cozens. Programming parrot. http://www.perl.com/pub/a/2001/04/01/parrot.htm, April 2001.

[JAM] The jamaica project. http://www.cs.manchester.ac.uk/apt/projects/jamaica/.

[Mat04] Richard George Matley. Native code execution within a jvm, 2004.

[MG00] Erik Meijer and John Gough. Technical overview of the common language runtime. http://research.microsoft.com/ emeijer/Papers/-CLR.pdf, 2000.

[Par] Parrot. http://parrotcode.org/.

[PBC] The parrot bytecode (pbc) format. http://www.parrotcode.org/docs/parrotbyte.html.

[PUG] (p)erl6 (u)ser's (g)olfing (s)ystem. http://pugscode.org/.

[RST04] Allison Randal, Dan Sugalski, and Leopold Ttsch. *Perl 6 and Parrot Essentials*. O'Reilly, 2nd edition, 2004.

[RVM] The jikes research virtual machine (rvm). http://jikesrvm.sourceforge.net/.

[SGBE05] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. In *Proceedings of*

the ACM/SIGPLAN Conference of Virtual Execution Environments (VEE 05), Chicago, Illinois, June 2005.

[Wal00] Larry Wall. State of the onion 2000. http://www.perl.com/pub/a/2000/10/23/soto2000.html, October 2000.

Appendix A

About the name "Parakeet"

It is common for projects related to Parrot to take names from other birds. For instance, the Ruby compiler targeting Parrot is named *Cardinal*, after a ruby-red bird found in north and south America.

The project should be named after a bird then, but preferably a bird that could somehow be related to Java. Enter "Parakeet". Parakeets are a sub-species of Parrots. Some Parakeets, like the Green Parakeet and the Orange-Fronted Parakeet, live in trees that shade coffee plantations. With Java being a kind of coffee, the link was made!

There is a downside to the name: so-called "coffee birds" are in danger of extinction, as increased demands for coffee has made farmers remove the shading trees to get more space for coffee bushes. With the habitat rapidly disappearing, the birds are not able to adapt fast enough and die out. It is to be hoped that such misfortune has not rubbed off on this project!