Optimizing Pinocchio

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Camillo Giovanni Bruni
26. Januar 2011

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Toon Verwaest
Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Camillo Giovanni Bruni
http://scg.unibe.ch/pinocchio

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Acknowledgments

I would like to thank my supervisor Toon Verwaest for his steady input of new ideas – Prof. Oscar Nierstrasz for his calm way of handling things – Bettina Gnägi for making me sleep at normal hours – Gabriel Jackson for eating lunch – Felix Bänteli for his musical knowledge – Migros Ice-Tea for keeping me awake – Anna Annen for making me study computer science – Cornelia Bruni for the stickers on my fridge – Mauro Bruni for teaching me Apple in the first place – my Brothers for their stupid jokes.

Abstract

Optimizations are an omnipresent topic when working on Virtual Machines (VMs). There is a plethora of different optimizations available ranging from simple tweaks and tricks to full evaluation concepts requiring a complex infrastructure. Depending on the complexity of an optimization and the performance increase it is important to choose the right kinds of optimizations. Based on a high-level language VM as a case study we argue in favor of transparent optimizations which do not require changes in the interpreted language's semantics. Furthermore it is necessary to write and properly evaluate benchmarks to be able to track the performance impact of a certain optimization.

When building a high-level language VM the underlying system – traditionally a C or C++ core – does not share many concepts with the implemented language. Hence some optimizations emerging from the low-level VM core are orthogonal to the high-level concepts of the interpreted language. Focusing on such optimizations can strongly limit the dynamic capabilities of a high-level language. These non-transparent optimizations require the semantics of the interpreted language to be changed. Changes in the language's semantics can require extensive changes in the sources which is an undesired property. However transparent optimizations preserve semantics of the language. Using transparent optimizations helps to separate the low-level requirements of the VM form the high-level design decisions of the language. We argue that non-transparent optimizations should only be applied to a high-level language VM in an early development stage. Furthermore each non-transparent optimization should be paired with a compatible way to reintroduce the lost or altered semantics.

To make valid statements about optimizations it is necessary to write adequate benchmarks. The benchmarks have to be reproducible and the evaluation has to be statistically sound, furthermore the benchmarks should focus on covering specific use cases to help locating performance issues in the VM code. Benchmarks are to be used in a similar way as unit tests. Optimizations can only be evaluated in a sound way when the corresponding benchmarks produce deterministic values. Furthermore we state that focusing on micro benchmarks helps to locate and track performance critical code segments of the VM.

Contents

1 Introduction **1**
- 1.1 Flexibility versus Speed . . . 2
- 1.2 Benchmarking . . . 2
- 1.3 Optimizations . . . 3
- 1.4 Outlook . . . 4

2 Pinocchio **5**
- 2.1 Pinocchio . . . 6
- 2.2 The Evolution of Pinocchio . . . 7
  - 2.2.1 SchemeTalk . . . 7
  - 2.2.2 Scheme Pinocchio . . . 8
  - 2.2.3 Smalltalk Pinocchio . . . 10
  - 2.2.4 Pinocchio with Opcodes . . . 11
- 2.3 Pinocchio's Metaobject Protocol . . . 12
  - 2.3.1 First-class AST nodes . . . 12
  - 2.3.2 Does not understand . . . 13
  - 2.3.3 First-class slots . . . 14
- 2.4 First-class Interpreters . . . 15
  - 2.4.1 Minimizing the Interpreter Stack . . . 15
  - 2.4.2 An example Interpreter: The Pinocchio Debugger . . . 16
  - 2.4.3 Extending the Debugger . . . 18
- 2.5 Summary . . . 21

3 Benchmarking **24**
- 3.1 Statistic Evaluation . . . 25
- 3.2 The PBenchmark Framework . . . 26
  - 3.2.1 Related Benchmarking Frameworks . . . 26
  - 3.2.2 Implementation Details of PBenchmark . . . 26
- 3.3 DTrace . . . 29
- 3.4 Summary . . . 30

4 Optimizations **31**
- 4.1 Type of Optimizations . . . 32
  - 4.1.1 Transparent Optimizations . . . 32
  - 4.1.2 Non-transparent Optimizations . . . 33
  - 4.1.3 Embedding Assumptions . . . 33
  - 4.1.4 Caching Data . . . 33
- 4.2 Default Benchmarks . . . 34
  - 4.2.1 Default Benchmark Results . . . 34
- 4.3 Native Dictionaries . . . 35
  - 4.3.1 Implementation . . . 36
  - 4.3.2 General Evaluation . . . 38
  - 4.3.3 Detailed Comparison of the Dictionary Implementations . . . 38
  - 4.3.4 Issues . . . 47
- 4.4 Opcodes . . . 49

4.4.1 Evaluation . . . 53
4.4.2 Issues . . . 56
4.5 Caching Integers and Characters . . . 57
4.5.1 Evaluation . . . 57
4.5.2 Issues . . . 59
4.6 Inline Caches . . . 59
4.6.1 Evaluation . . . 61
4.6.2 Issues . . . 63
4.7 Unwrapping Values . . . 64
4.7.1 Evaluation . . . 66
4.7.2 Issues . . . 66
4.8 Future Work . . . 67
4.8.1 Compiler . . . 68
4.8.2 Runtime Optimizations . . . 69
4.8.3 VM Optimizations . . . 69

5 Conclusion **71**
5.1 High-level Languages and Optimizations . . . 72
5.2 Make it Work, Make it Right, Make it Fast . . . 73

Appendices **77**
A.1 Default Benchmarks Sources . . . 77
A.1.1 Dictionary . . . 77
A.1.2 Parser . . . 78
A.1.3 Fibonacci . . . 80

1 Introduction

Developing high-level language virtual machines (VMs) is generally a balancing act between the flexibility and the speed of the resulting system. On the one hand a system can be more performant if more and more assumptions are hard-coded in the VM itself but might reduce the language's dynamic capabilities. On the other hand creating a more flexible VM requires more indirections while executing code and thus might have negative speed impact. However by using the right optimization techniques it is possible to reduce the number of indirections at runtime.

Choosing the right optimization depends mainly on two factors: the implementation complexity and the performance boost achieved. Next to the direct implementation costs of the optimizations a secondary cost can be observed. Depending on the optimization it is necessary to adapt the interpreted language. Such non-transparent optimizations are generally unwanted. Changing a language's semantics can require significant changes to the sources and adaption of existing unit tests. By observing the Pinocchio VM – a Smalltalk-like system – we see that most of the optimizations applied were transparent. Non-transparent optimizations have only been applied in the early stage of development. With a more stable system comes more responsibility – optimizations have to focus on preserving the existing semantics in order to avoid secondary implementation costs. However, as observed in Pinocchio, a well defined non-transparent optimization can give a signification performance boost. Generally it is possible to create a compatible way to make the lost or altered semantics accessible again. Although this way of reintroducing lost features due to optimizations does not relieve the programmer from adapting the existing sources, it is possible to reduce the amount of changes necessary to render the code compatible again

It is not always possible to make the right choice when applying certain optimizations . Thus it is important to track the impact of each applied optimization, since it might happen that a new optimization has a negative impact on the overall system. This requires a similar mechanism as regression tests or unit tests. With proper benchmarking it is possible to track the speed impact of each optimization in-depth. Benchmarks should be written specifically to track single optimization properties, similar to unit test preferably focusing on the smallest behavior possible. Using micro benchmarks helps to locate performance critical code segments of the VM implementation. In contrast general benchmarks focusing on common language use cases provide a convenient way to compare the overall performance of a system. But with such a single result it is almost never possible to identify a certain bottleneck of a VM. In order to optimize a VM

it is necessary to write specific micro benchmarks to measure the impact of a specific optimization. For Pinocchio a set of three benchmark suites consisting of several micro benchmarks is used to identify potential issues with the different optimizations applied.

1.1 Flexibility versus Speed

Implementing high-level language VMs requires to choose between speed and dynamic features of the interpreted language. Generally it is possible to write a fast VM by altering the design of the interpreted language to follow more closely the low-level implementation language – generally C. On the other hand, a very flexible but unoptimized system requires more indirections during execution. Furthermore the mismatch between the VM definition language – generally C – and the high-level language being interpreted can introduce an additional overhead.

As an example of such a dynamic system we take SchemeTalk – a Scheme dialect with the semantics of Smalltalk. In SchemeTalk the semantics of Smalltalk are introduced to the underlying Scheme by a set of macros. Since the interpreted language – a Scheme dialect close to Smalltalk– and the VM definition language – also a Scheme – are syntactically and semantically very similar there is no clear boundary between the VM definition and the interpreted language. It is noteworthy that SchemeTalk does not feature a real VM implementation but rather consists of a set of macros to transparently add the semantics of Smalltalk. In fact objects can freely flow between the Smalltalk world and the Scheme world. Furthermore objects from the Smalltalk world can only be accessed in a well-defined manner from the Scheme level. This is only possible since Scheme has many features built into the language which can be reused for running a Smalltalk-like language on top. For instance to build the Smalltalk objects SchemeTalk uses Scheme level lambdas which already encapsulate data and behavior. Hence it is not possible to break the encapsulation of the Smalltalk objects by passing them down to Scheme and directly accessing the raw values stored in the object. The advantages of a compact and simple VM definition are traded for a rather slow system. The resulting system is roughly 1000 times slower than the original Pharo system.

The question arises on how to optimize such a system while keeping its dynamic nature. In a first attempt we tried to avoid the scheme interpreter by implementing a low-level core system in C. The resulting system featured AST nodes which knew how to evaluate themselves. This way the system can be extended with arbitrary new AST nodes since the VM itself expects only a very limited interface to activate the nodes. The resulting system was even more flexible than its predecessor SchemeTalk and in this case again did not meet the performance expectations. The self-evaluating AST nodes introduced too many indirections. From here on a plethora of optimizations were applied to the system and eventually resulted in Pinocchio with a rather classical VM design with a performance in the range of common scripting languages like Python, Ruby or Smalltalk.

1.2 Benchmarking

In order to improve the speed of VM it is important to have proper benchmarks tracking different performance aspects. However there are several important properties of benchmarks which are ignored too frequently. The three core aspects of benchmarking which drive the evaluation of this thesis:

Reproducible results: Benchmarks should work similar to unit tests and have to be reproducible with a minimal effort.

Micro benchmarks: Focusing on micro benchmarks helps to identify performance critical sections of a high-level language VMs. This is again similar to unit tests which help to isolate bugs by writing tests for single features.

Statistical sound evaluation: The results from the benchmark runs have to be presented with respect to a basic statistical evaluation.

The first two principles, reproducibility and micro benchmarks, are also key elements of unit testing. Unit tests have to be reproducible in order to make sense. If a test provides non-deterministic results it is of no use. The same principle counts for benchmarks: they only provide useful information if the results can be reproduced by different programmers.

By writing small tests which cover only a limited set of functionality, the task to locate a faulty piece of code is greatly simplified. Again, the same holds for benchmarks. By writing an overall benchmark, it is possible to provide a simple results to track the performance of a whole system. However, this is only of limited help if the goal is to improve the performance of a VM. The overall benchmark result hides too many details. Hence writing micro benchmarks helps to spot performance critical code segments.

The third principle is new, compared to the unit test approach. Unit tests ultimately provide only a single boolean results, either the test succeeded or failed. However, a benchmark results is calculated from an average of several timed benchmark runs which requires a minimal statistical evaluation. Although it would be possible to use the time spent of only a single run, it would provide only a very inaccurate result. The time spent for a benchmark relies on a multitude of parameters of the host system, controlling all of them is almost impossible. For instance each running background process, or the overall memory consumption of the host system might impact the outcome of a benchmark run. Hence the most common way to eliminate these error sources is to run a benchmark several times and thus average out the different system setups at different times.

The straight-forward way to address these three principles is by writing a simple benchmarking framework. Similar to a unit testing framework, the programmer can focus on writing the single benchmarks and does not have to deal with the low-level details of benchmark evaluation. For this thesis a simple benchmarking framework called `PBenchmark` was developed in Smalltalk and has been used to evaluate a set of optimizations. The details of the benchmarking framework and the limited statistic background needed to evaluate benchmark results are presented in Chapter 3.

1.3 Optimizations

By observing the high-level language VM Pinocchio we identify a broad classifications for optimizations:

Transparent optimizations focus on optimizations which do not enforce semantic changes on the interpreted language. A typical example of such an optimization is a just in time compiler which optimizes the underlying evaluation structure at runtime.

Non-transparent optimizations require the semantics of the interpreted language to be altered. An example of such an optimization is the switch from an untyped to a typed language.

Embedding assumptions: This group of optimizations focuses on making the low-level VM implementation aware of the high-level structures used in the interpreted language. By embedding the assumptions for common use cases into the

VM execution speed can be greatly improved. The goal is to reduce the overhead introduced by the semantic mismatch of the VM definition language – generally C – and the interpreted language. A common example is the replication of high-level data structures on the C level.

Caching values focuses on avoiding the repeated calculation of the same values. An example for frequently used caches are inline caches.

The four groups of optimizations are not mutually exclusive and the transparency property in particular overlaps with the last two properties. The last two properties, embedding assumptions and caching values, are a classification on the low-level implementation details of an optimization. The transparency property, however, describes a more abstract class of optimizations.

In this thesis, optimizations from all four groups are presented, however, only a set of four non-transparent optimizations is presented in more detail. Non-transparent optimizations are generally applied in the early stages of a VM. Significant changes in semantics are to be avoided, especially in a more mature system. In Pinocchio several non-transparent optimizations were applied in early development versions of the VM as described in Chapter 2. At that point the code base was still very limited and thus semantic changes could be easily reflected onto the existing sources. In the later stages, having already a rather mature system, optimizations were mainly transparent. This allowed us to separate the low-level work on the VM from the high-level design decisions. This is especially important in a high-level system which is semantically distant from the VM definition language. Otherwise, it could happen that many features of the VM definition language are adopted for performance reasons. Hence we argue that the focus should lie on transparent optimizations, such as those presented in more detail in Chapter 4.

1.4 Outlook

The rest of this introduction identifies important properties of VM implementations, benchmarking and the choice of optimizations. In the rest of this thesis the most important design and optimization decisions taken in the Pinocchio VM are presented in more detail. Throughout the design process we paid attention to keep the VM extendible but still were able to achieve a speedup of around two orders of magnitudes. In Chapter 2 Pinocchio and its predecessors are presented giving a broad overview of the design choices and a limited set of optimizations applied. The core of all these optimizations is to embed assumptions to avoid as many indirections as possible in the low-level code. Nevertheless it is possible to apply most optimizations such that there is backwards compatibility. As an example of such behavior we present the C implementation of the common dictionary data structure in Pinocchio in Section 4.3. Five other important optimizations are presented in Chapter 4 ranging from familiar inline caches to a Pinocchio specific dictionary implementation. Each optimization is explained with the results of three different benchmarks. Chapter 3 provides reduced statistical background information needed to perform a sound benchmark evaluation. The same chapter provides an overview of our custom benchmarking framework which provide reproducible results. In Chapter 5 we summarize the results and conclude.

2
Pinocchio

In order to distinguish between different types of optimizations we target the Pinocchio VM. Pinocchio is a Smalltalk-like high-level language VM used in this thesis to show different aspects of optimizations. Up to the version discussed in this thesis, Pinocchio has undergone major changes. Its predecessor is a Scheme dialect with Smalltalk semantics called SchemeTalk. By applying taking radical optimization decisions the system evolved into a more classical VM with an opcode based interpreter. Looking at the history of Pinocchio we can identify several non-transparent optimizations in the early stages whereas most optimizations which happened later are transparent.

The first sections provides a broad overview of the features of Pinocchio. The origins of most of the current features are then described in the next section covering the evolution of Pinocchio and its predecessor SchemeTalk. The last section covers in more detail the manifold extension mechanism of Pinocchio and sheds light on possible optimization issues.

2.1 Pinocchio

The different optimizations in this thesis are presented by observing the high-level language VM Pinocchio. Pinocchio is a modern VM for a Smalltalk-80 [4] dialect with focus on extendability. Additionally to the already strong reflective capabilities of Smalltalk Pinocchio adds first-class interpreters and first-class slots to the language. Interpreters can be created, run and extended at runtime. Customized interpreters evaluate AST nodes and can change the default evaluation by overwriting visitor methods. Unlike the custom interpreters the core interpreter is based on the much faster opcode evaluation scheme.

To construct a new variant of the Pinocchio interpreter it suffices to subclass the `Interpreter` class and override a part of its visitor interface (see Figure 2.1). The `Interpreter` class defines a meta-circular interpreter implemented as an AST visitor that manages its own environment but relies on recursion to automatically manage the runtime stack.

The meta-circular interpreter reifies the core interpreter written in C. Eventually the meta-circular interpreter will use natives for its methods which hook into the underlying C interpreter code. However so far this has never been realized and is impossible to achieve with the current opcode based evaluation scheme. Hence the meta-circular interpreter is almost fully written in Pinocchio itself. However there is one exception when a real native needs to be evaluated. In this case the meta-circular interpreter has to pass on control to the underlying interpreter for native evaluation. This example is demonstrated in the following code excerpt.

```
invokeNativeMethod: aClosure on: receiver
   message: aMessage alternative: aBlock
   <pPrimitive: #invokeNativeMethod:on:message:alternative:
         plugin: #'Interpretation.Interpreter'>
   ↑ aBlock value
```

The first line of the method body activates the evaluation of the native denoted by `aMessage`. If this should fail the block passed on in the argument `aBlock` is evaluated as an alternative.

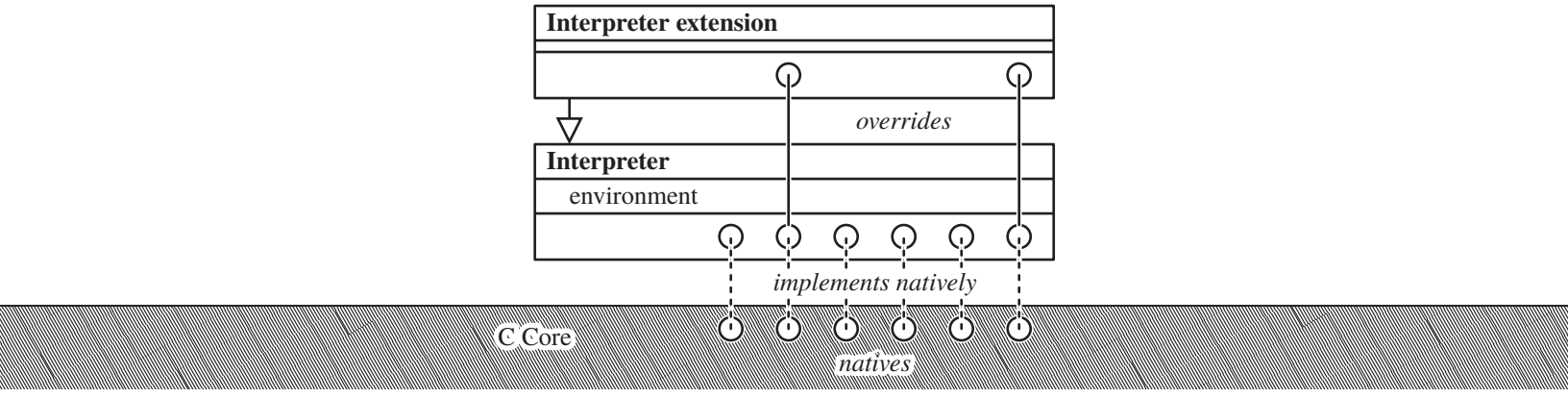

Figure 2.1: Native methods in the `MainInterpreter` and interpreter extension through subclassing

Application code is evaluated by a new interpreter by sending the `interpret:` message to the desired interpreter class with a closure representing the code as its argument. For example, the expression

```
Debugger interpret: [ self runApplication ].
```

will cause the closure `[ self runApplication ]` to be evaluated by the `Debugger` interpreter. This special interpreter is further described in Section 2.4.2.

As usual, closures encapsulate an *environment* and an *expression* object. When passing the block `[ self runApplication ]` to the specialized interpreter the environment captured by the block is installed in the interpreter. The expressions are then evaluated in the environment of captured by the block. Since the passed expression for the default interpreter is a closure the expression is evaluated by sending the message `value` to the closure in first-class the interpreter. The following snippet shows how the closure argument is evaluated by sending the `value` message at the base level:

```
interpret: aClosure
   "base level block activation"
   ↑ self send: (Message new selector: #value)
         to: aClosure.
```

Although it might seem correct to directly evaluate the closure by invoking `value` on it, this is incorrect as the closure would be evaluated at the wrong level of interpretation. It would run at the level of the interpreter (the meta-level from the application's point of view) rather than at the application level as desired. Another issue with the direct invocation is that the MOP for message activation would be surpassed.

The open design of the meta-circular interpreter lets programmers extend the runtime with very little effort. More importantly, the extensions to the interpreter are implemented within the language provided by the interpreter itself. As such they can be implemented using any of the existing tools for the language, including development environments, debuggers, test runners and versioning systems. Further details about customized interpreters are given in Section 2.4.

2.2 The Evolution of Pinocchio

This section covers the different stages of the Pinocchio interpreter, from its predecessor SchemeTalk to the first version of Pinocchio supporting a Scheme, on to the current version.

2.2.1 SchemeTalk

SchemeTalk can be seen as the main origin of Pinocchio. SchemeTalk was developed by Toon Verwaest to examine the properties of a highly dynamic language. SchemeTalk is an object-oriented language built on top of Scheme. It combines the syntax of Scheme with the message send semantics of a Smalltalk system. Objects are implemented as dispatch objects on top of Scheme. Given the fact that the language of the VM and the evaluated one are very alike the interface of the VM could be built with a unified interface. Verwaest and Renggli [12] describe that this is an important property for simplifying the creation of new programming languages on top of an existing VM. In a hybrid system the encapsulation of objects is not guaranteed when crossing the barrier from the application level to the VM level. Typically the VM can directly access the application level objects. In combination with reflective features that need support from the underlying VM level to access certain properties this poses a certain risk. Reflection in combination with the raw VM level access can be used to invalidate or manually alter objects. This is not possible since the application-VM barrier would preserve encapsulation. Generally this happens in languages abstracting away many low-level features which might be needed for the new language. In this case the VM can be opened using reflection [8] to manually inject new features. But since the reflection breaks encapsulation the VM has no knowledge about the injected code.

An example of such a feature [7] is backtracking in Smalltalk without direct support from the underlying VM. The only way to change execution in Smalltalk is to manually

realign the stack frames. This is problematic in the case of an optimization that is based on the assumption of a valid VM state which achieved only through encapsulation preserving operations. In the case of manually changing the stack the optimization has to be fully aware of that. For instance a tracing JIT will directly rely on the C stack for speed fully omitting the use of the VM-level stack. Each time the stack is manually managed in the application code the stack has to be reconstructed for the JITed parts.

In SchemeTalk the VM is a homogeneous system in which encapsulation cannot be broken. Application level objects are constructed using closures which are not introspectable from within the Scheme code. Due to its open design, SchemeTalk exposes almost all types of low-level behavior in application-level code. Hence new language features in SchemeTalk can be expressed from within the application level code. This further implies that new language will not break encapsulation by accessing VM internal features.

Next to the fact that the VM and the interpreted language are conceptually very close there are further interesting concepts in SchemeTalk which are notable and partly influenced Pinocchio. As extensions to a standard Smalltalk implementation SchemeTalk supports class methods, a well-defined MOP for method activation and first-class slots. Slot accesses are handled by always following the MOP and sending getter and setter messages to the slot objects. Method activation is handled in a similar way. First the method is looked up in the class hierarchy much like in a traditional Smalltalk implementation. The found method is activated using an extended MOP compared to what is available in a Smalltalk system like Pharo. Generally the found method is passed down to the VM for activation leaving no possibility for changes from within Smalltalk. The only MOP available for method activation in an already compiled Smalltalk method is during the lookup by using the `doesNotUnderstand:` hook whereas in SchemeTalk methods are activated upon receiving an `execute` message. This way special method objects can be used with changed activation properties. It is possible to emulate the behavior for changing the method activation by using the `perform:with:` MOP of Pharo. However, this requires the code to be explicitly changed in a very verbose manner. Further extensions to methods are meta methods. Similar to Python's `@` decorators they allow a method to be wrapped in a meta method adding additional behavior.

The drawback of this elegantly designed and highly flexible system is performance. Each method activation and slot access is slowed down by several indirections of message sends. The goal of the succeeding version of Pinocchio is to avoid these indirections for the most common cases to increase speed but still maintain a comparably high level of flexibility.

2.2.2 Scheme Pinocchio

The first version of Pinocchio emerged from the SchemeTalk project. It addressed performance issues by a reimplementation of the core system in C instead of Scheme. Nevertheless it introduced another level of flexibility by encapsulation the evaluation of each AST node directly on the objects themselves and not just centralized in the VM core. This way the existing MOP for method activation and slot accesses of SchemeTalk could be absorbed like most of the data structures. The language executed on top of the first Pinocchio version was Scheme. The choice was driven by the comparably simple compiler needed to support the language. Since in SchemeTalk the evaluated language and that of the VM are very close – and even the same on a syntactical level – the core system is rather small. The C version needed some work to achieve the same state and involved for instance the implementation of method context – something you get for free when implementing the VM on top of Scheme.

By dropping some of the flexible properties of SchemeTalk a different concept of how to dynamically modify the code execution was explored. Pinocchio complemented

the self evaluating AST nodes with first-class interpreters. The idea behind this is to provide custom tailored execution next to a fast base system which knows how to directly evaluate common AST nodes. The approach used in SchemeTalk enforces many indirections in the core system, whereas the goal with the interpreter approach is to only introduce indirections when needed. Ideally a new interpreter is only launched for a limited time when special execution behavior is needed, for example a debugger. Debugging generally involves programmer interaction on each message send for instance to allow the programmer to step through the code. To do so the running interpreter needs to support these kinds of stepping actions. However it would be wasteful to do part of this even in normal execution mode. Whereas a traditional VM needs a special hard-coded hook to support debugging, in Pinocchio you would launch a debugger interpreter. The debugger shares most of the code with the default implementation but adds an indirection upon each message send. Once debugging is finished and the control flow leaves the debugger interpreter, execution resumes to the normal mode.

From a developer point of view there are several pros and cons to be mentioned. Compiling the core system for instance, involved exporting C code that builds up the AST nodes. This worked with a simple compiler outputting in a verbose fashion very simple C statements. Hence at that point module based compilation was not possible – only the whole system could be compiled at once. Compared to the next version of Pinocchio the system was still rather small, thus the single file approach was not that problematic. However locating errors was more cumbersome as it is not directly visible to which class or method a certain C statement belongs. However a nice fact about this version of Pinocchio is that natives could be directly written with inlined Scheme code. The following code illustrates this feature:

```
1  ((eval ()
2     "
       call_object ast_call = context->self.ast_call;
4      object env = context->env;
       context->code = &ast_call_invoke_env;
6      "
       (send ast_call->target eval)
8      "
       inc();
10     ")
```

On line 7 the application-level message `eval` is sent to the object at the C location `ast_call->target`. The inlined code would then be fully expanded by the compiler to C code which builds up the corresponding AST nodes. Thus the native code was rather compact and only the crucial parts had to be written in C avoiding the cumbersome manual creation of AST nodes. A rather unlucky design choice was the registration of new natives in the core system. Adding a new native involved changing sources in 3 different locations.

Another exceptional mechanism to be mentioned is how arguments are handled in the first version of Pinocchio. The Scheme version featured call-by-reference which is not possible anymore in the Smalltalk version of Pinocchio. Rather than letting the VM evaluate arguments directly the raw expressions were passed to the method activation. Each method had to evaluate the arguments on its own. Although certain shortcuts are provided the main responsibility stayed on the method body itself. The methods in Scheme Pinocchio are split up into two distinct parts. The first part handles the argument evaluation in the old context (the call site) and a second part creating the new context and evaluating the method body in it. This was changed later on in the Smalltalk version of Pinocchio where argument evaluation is directly handled by the VM. Conceptually this was a nice feature to have but introduced additional overhead in most cases. Since in the general case there is no need to interfere with the method

evaluation. Thus the default case was hard-coded in the VM.

Implementation-wise the first Pinocchio VM had several features which took a long time to reappear in the following versions. The most important one is that the Scheme version could handle exceptions quite early thus allowing for writing proper test cases. This allowed us to write test cases directly in Scheme code and properly handle assertions. In the following Smalltalk version, Pinocchio only C level assertions could be used for a long time. This is clearly visible in the lack of unit tests for the core system.

2.2.3 Smalltalk Pinocchio

After implementing a working Scheme on top of the first Pinocchio VM we decided to implement a second language interface for Smalltalk. However the syntax of Smalltalk requires a more elaborate parser and compiler than for Scheme. During the development of the Smalltalk the shortcomings of the Scheme Pinocchio became obvious. A too flexible system has not only advantages and in this case might even slow down the development. The main drawbacks already mentioned in the previous subsection are the registration mechanism for new classes spread over 3 different source locations and a complex argument evaluation mechanism. Hence we started already in parallel on a second version of Pinocchio to overcome the speed limitations and the design issues of the Scheme version. The main difference to the Scheme version are how the ASTs are evaluated. Instead of directly allowing special behavior to be installed on the AST nodes themselves we hard coded the most common cases. The following excerpt shows the main dispatch for AST evaluation.

```
void send_eval(Object exp, Type_Class class, Array args)
{
   EVAL_IF(AST_Assign)
   EVAL_IF(AST_Constant)
   EVAL_IF(AST_Variable)
   EVAL_IF(AST_Self)
   EVAL_IF(AST_Block)
   EVAL_IF(Slot_Slot)
   EVAL_IF(Slot_UIntSlot)
   EVAL_IF(Organization_ClassReference)

   //MOP: visit the AST node with the current interpreter
   Type_Class_direct_dispatch(exp, class,
                         SMB_accept_, args);
}
```

The `EVAL_IF` macros expand to simple type checks and a call to the corresponding AST evaluation function in C. The expanded version of `EVAL_IF(AST_Assign)` would look like the following snippet:

```
if (class == AST_Assign) {
   AST_Assign_eval(exp, args);
   return;
}
```

The last statement in the larger example above is the fallback mode for unknown or customized AST objects. It activates the AST node using the current interpreter – an AST visitor – by sending the `accept:` message with the current AST node as argument. Since the behavior of certain AST nodes is known upfront the evaluation speed can be significantly improved by shortcutting the AST visitor/interpreter and directly calling the proper C function. Hence we avoid several layers of indirection introduced by the AST visitor/interpreter but lose some flexibility as we can no longer change the behavior of the core AST nodes. Custom AST nodes are still supported by writing a

custom AST visitor with additional visitor methods for the custom node types. However this approach clearly limits the capability of the main interpreter compared to the previous version of Pinocchio. New kinds of AST nodes can no longer be introduced at any point in time but are coupled to a specific type of interpreter. However by storing the evaluation behavior of the AST nodes centralized in an interpreter the general evaluation performance is greatly improved.

Early versions of the Smalltalk version still supported custom AST nodes through a special MOP. The default visitor could send the `eval` message to unknown AST nodes and thus fully support the old evaluation scheme and still be fast in the common cases. In later versions of Pinocchio the support for custom AST nodes was dropped due to introduction of opcodes. A possible solution for custom AST nodes is based on the idea of using the nodes themselves as code generators. The custom AST nodes could provide a template for the visitor methods in the interpreters. This way the behavior of all AST nodes, custom and known, could be defined directly on the nodes without a secondary or parallel implementation in an interpreter.

Even though the Smalltalk version is already much faster than its Scheme-based predecessor it was still around 10 times slower than a comparable Smalltalk implementation like Pharo. The visitor-based AST evaluation is expensive as it requires many unnecessary stack operations to evaluation the tree-based structure. The next version of Pinocchio explicitly addresses this issue by using opcodes as main interpretation structure which flattens out the indirections introduced by the AST nodes.

2.2.4 Pinocchio with Opcodes

During a phase where the progress of Pinocchio was focused on performance we tried to improve speed by creating more and more native methods. The most common example where this approach works well is the dictionaries since the public interface of the class is minimal. Thus a great part can be implemented in C resulting in significant speedups. But we also tried to improve performance for instance on the boolean class by directly implementing native versions of `ifTrue:` or `ifFalse:`. This comes with mainly two disadvantages. Firstly we have to program in C and may abandon a bit the dynamic world of Smalltalk and secondly we have two versions of the same code. Resuming the example of dictionaries, on several occasions the C version did not incorporate the newest design choices made in the Smalltalk version. This was mostly due to the imposed overhead of programming and debugging in C. These examples should strengthen the argument that natives are only useful for either very performance critical libraries or general data structures with a substantial part of the functionality being hidden behind a simple interface. Focusing on performance we brought much attention on the evaluation of ASTs. Of course evaluating the ASTs directly with a visitor has significant overhead. Mostly due to the treelike nature of ASTs, evaluation cannot be directly flattened out and requires temporary values to complete evaluation. As an example we take the simple binary message send +. First the left-hand side is evaluated and pushed on the stack followed by the same operation for the right-hand side. Eventually the two arguments are popped from the stack, summed up and the sum is pushed again on the stack for further usage. Generally speaking there are always multiple stack operations necessary to keep track of the current position within the AST evaluation. However this is not the case with bytecodes as the control flow has already been flattened as for instance in loops. Loops can be performed by directly jumping into a list of statements whereas the same behavior using pure ASTs involves many block activations and superfluous stack operations.

Hence as the performance increase was stalled and no further optimization was in sight we decided to build an opcode compiler and evaluator. It is important to say that Pinocchio does not use bytecodes as the only evaluation structure. Rather we focused on a dual system that is compatible with the existing AST visitor. This is achieved by

keeping both data structures alive next to each other. Each method contains the original ASTs – used for inspection and refactoring – and a list of opcodes and values for fast evaluation. A second compiler step transforms the original ASTs into opcodes. For instance a common send AST is transformed into a send opcode with the original send object in the next position. When the send opcode is invoked it takes the send AST out of the next position and evaluates it in the usual way. Furthermore we optimize special boolean messages like `ifTrue:` or looping constructs like `to:by:do:` where opcode evaluation outperforms AST evaluation by far. At the moment the following send instructions are optimized:

- `ifTrue:`
- `ifFalse:`
- `ifTrue:ifFalse:`
- `ifFalse:ifTrue:`
- `to:by:do:`
- `to:do:`

By using special opcodes for the most common sends and constants the number of natives could be drastically shrunken. The native methods mentioned previously for the booleans are no longer needed. Even though we removed some natives the opcode execution is still significantly faster. Directly traversing a list is much faster than following the indirections introduced by an AST visitor and avoids multiple stack operations to track the current location in the AST tree. By using opcodes and certain simple optimizations we could achieve an overall speedup of a factor 10. The specific speedups for the different optimizations used with opcodes is discussed on page Section 4.1.3.

Pinocchio has been built to provide a flexible high-level language runtime which is extendable by first-class interpreters and a well-defined MOP. This section covers the separate extension mechanisms in more detail. The first subsection covers the MOP of Pinocchio which is for the most part what Smalltalk does. The second part covers the creation of new interpreters.

It is important to list these different extension mechanisms separately in order to point out several optimization issues. Generally the more flexibility is added to a language the more indirections during evaluation are introduced. This can affect the performance of such a high-level system in a negative way.

2.3 Pinocchio's Metaobject Protocol

Aside from providing access to user-definable and first-class interpreters, Pinocchio provides a default metaobject protocol that is sufficient for many reflective use cases. From the point of view of the programmer, the main interpreter is written as a metacircular AST visitor whose methods can be overwritten to change standard evaluation. Another way to add new semantics to the language is by replacing standard application constructs such as methods with custom metaobjects following the same metaobject protocol. The following extension points are noteworthy.

2.3.1 First-class AST nodes

New nodes can be defined by following the visitor protocol. The new nodes can be generated by extending the default parser and compiler. This can for example be used to provide mutable AST nodes or *link* objects for partial behavioral reflection. As a second step additional visitor methods can be provided in a custom interpreter class to deal with the new AST nodes. The following output shows a live inspection in a running Pinocchio to give an overview of the AST features:

```
> methodClosure := 1 class methodDict at: #'-'
MethodClosure
      code: NativeMethod
      selector: #'-'
      host: Kernel.Number.SmallInteger

> method := methodClosure code
NativeMethod
      params: Array
      locals: Array
      package: Nil
      annotations: Array
      info: Info
      threaded: ThreadedCode
      1: Send
      2: Self

> method annotations first
Annotation
      selector: #'pPrimitive:plugin:'
      1: #'-'
      2: #'Number.SmallInt'

> method at: 1
Send
      cache: InlineCache
      message: #'pinocchioPrimitiveFailed'
      receiver: Self

> method annotations
Array
      1: Annotation
```

AST nodes were subject to several bigger optimizations during the development of Pinocchio. Initially the AST nodes encapsulated the full evaluation behavior. From an object-oriented point of view this was a very elegant solution in SchemeTalk. However this design introduced too much overhead. The next versions of Pinocchio directly hard-coded the behavior for the most common AST nodes thus avoiding most of the indirections.

2.3.2 Does not understand

Following Smalltalk-80, our core interpreter sends the `doesNotUnderstand:` message to any object that does not implement a method corresponding to the selector of a message sent to it. This is an important feature to make Pinocchio compatible with existing Smalltalk code.

```
Interpreter>>send: aMessage to: receiver class: class
   | method |
   method := self lookupSelector: aMessage selector
               in: class.
   method ifNil: [
      ↑ self sendDoesNotUnderstandFor: aMessage
           to: receiver ].
   ↑ method
      accept: self
```

```
      on: receiver
      message: aMessage
```

2.3.3 First-class slots

Unlike most Smalltalk systems which rely on *magic numbers* to encode the layout of instances, Pinocchio's class layouts are described using *layout* metaobjects. Magic-numbers used in Smalltalk are a bad unification of the structures used internally in the VM. Since there is almost a one-to-one mapping first-class objects could be used as well. Especially in a system that claims that everything is an object it is unfavorable to draw the line between high-level objects and VM internal values already at that level. Pinocchio shows that is possible to provide first-class objects even for the layout. The following session gives an example of inspecting the layout of a `SmallInt`, a `String`, an `Array` and a `Class`:

```
> 1 class layout
IntLayout
> 'a' class layout
WordsLayout
> Collection.Array layout
ArrayLayout ()
> 1 class class layout
ObjectLayout
      1: Slot(layout)
      2: Slot(superclass)
      3: Slot(methods)
      4: Slot(name)
      5: Slot(package)
```

These metaobjects further rely on *slot* metaobjects that define the semantics of instance variables. Whenever a reference to an instance variable is made in the application's source code the compiler directly inserts the corresponding slot metaobject into the resulting method's AST. The following session shows how to access the first-class slots from the `Dictionary` layout:

```
> Collection.Dictionary layout
ObjectLayout
      1: Slot(size)
      2: Slot(maxLinear)
      3: Slot(ratio)
      4: Slot(buckets)
      5: Slot(linear)
> Collection.Dictionary layout at: 1
Slot(size)
      index: 0
      name: #'size'
      package: Nil
```

Every slot metaobject can override the AST node evaluation protocol to provide custom semantics for retrieving the instance variable. Custom semantics for the assignment to instance variables are implemented by overriding the protocol provided by the interpretation of the `Assign` AST node. The default implementation of this protocol on `Slot` is given in the following example:

```
Slot>>assign: value on: anObject
   <pPrimitive: #assign:on: plugin: #'Slot.Slot'>
   self pinocchioPrimitiveFailed
```

```
Slot>>readFrom: anObject
   <pPrimitive: #readFrom: plugin: #'Slot.Slot'>
   self pinocchioPrimitiveFailed
```

The default implementation relies on natives to directly access the values stored by the slot object.

By providing explicit layout and slot metaobjects, applications can easily decide what kinds of layout and accessing semantics to attach to specific classes. Special behavior such as first-class relationships or singletons can cleanly be factored out into slot libraries to avoid cluttering of the code. Instead of repeating code segments – for instance for the singleton pattern – a special slot class can be reused.

This slot only instantiates an object on the first access and returns the same object from this point on. Of course to fully stick with the singleton pattern writing to this slot has to be prohibited by overwriting the corresponding `assign:on:` method.

At the moment slots are only used as descriptors in the layout and don't add behavior at runtime. However, future versions of Pinocchio should provide a library of reusable slot implementations. Similar to the use of natives, the performance of the default implementations should stay untouched and only the override versions should be evaluated meta-circularly.

2.4 First-class Interpreters

The MOPs presented in the previous Section 2.3 are quite standard. Although in current Smalltalk implementations only the Does-not-understand MOP is available. Smalltalk features first-class AST nodes but they are not used as an interpretation source but mostly for refactoring. First-class slots are also a rare feature in most modern high-level languages but a widely known concept in the language community [11]. But next to the three extension mechanisms presented earlier Pinocchio provides access to the evaluation by using first-class interpreters. The interpreters are AST visitors which allow for customization of most of the aspects of AST evaluation. Since the interpreters are first-class they can be treated like any other object, subclassing and runtime instantiation can be used normally. First-class interpreters emerged from its predecessor Scheme Pinocchio which featured self-evaluating AST nodes. In Scheme Pinocchio it is possible to introduce new types of AST nodes with different evaluation behavior into a running system. There was no interpreter controlling the evaluation. However this design introduced too much overhead and eventually was replaced by an interpreter based approach.

Although there is a core interpreter implemented in C it can be extended at runtime by subclassing. These first-class interpreters provide an interface to the evaluation of all the AST nodes by using the visitor pattern. By subclassing it is possible to introduce evaluation behavior for different kinds of AST nodes by having custom visitor methods. Thus the interpreter-based approach is as flexible as the original version with self-evaluating AST nodes. However instead of supporting any type of AST nodes in the general case Pinocchio only knows a certain set of AST nodes at a time. In the default case these are the nodes produced by the Pinocchio compiler. If a different set of AST nodes should be supported a new customized interpreter has to be instantiated and run on top of the existing one. This way each newly launched interpreter uses the interpreter below for its evaluation.

2.4.1 Minimizing the Interpreter Stack

The approach of starting new interpreters on top of other interpreters is similar to, albeit the inverse of, the tower of first-class interpreters in Refci [10]. This has the advantage

that we can use the same approach to minimize the height of the tower that is actually running at each point in time. For example, since the `MainInterpreter` does not alter the interpretation behavior of the standard interpreter, the evaluation of the application can happen fully at the C level, dropping the `MainInterpreter` from the active interpreter stack. It is important to never run on a stack bigger than necessary, since each extra level of interpretation has a steep price in terms of performance. For instance the `Debugger` interpreter introduces an overhead of around 6 message sends for each normal message send in Pinocchio. Thus running two debuggers on top of each other is already 36 times slower than the default interpreter.

In Pinocchio the height of the tower is pragmatically minimized by making the whole definition of the standard interpreter available as a fine-grained set of natives installed on the `Interpreter` class (see Figure 2.1). Only the extensions – the override visitor methods – to the interpreter are evaluated meta-circularly.

Since natives are able to send messages back to the application level, every call to `invokeNative` has to store the interpreter that triggered the actual native. Each time a native wants to send back a message on the application level the original stack of interpreters has to be restored. An example of such a case is the native implementation of the `at:` method installed on dictionaries. This method needs to be able to request the `hash` value of a key, and later compare it with the keys in the dictionary using the = message. Both methods are within the control flow of the native evaluation of the `at:` method but need to be evaluated on the application level. To evaluate both methods at the right level of interpretation, the stack of interpreters that was active before the `at:` was invoked needs to be reconstructed before the evaluation of each contained application level message is started.

2.4.2 An example Interpreter: The Pinocchio Debugger

So far the capability of Pinocchio to create new kinds of interpreters has only been discussed in a theoretical manner. This section covers in more detail the implementation of the Pinocchio debugger – a customized interpreter. Pinocchio does not feature a special VM mode for debugging but rather takes advantage of the first-class interpreters. Remember that in Pinocchio debugging is enabled by passing a statement to the `Debugger`'s `interpret:` method – as shown in the following statement:

```
Debugger interpret: [ 'a' + 1 ]
```

By sending `interpret` to the `Debugger` class a new Debugger is instantiated and the control is passed on to this new interpreter. As mentioned before, the passed-in block captures an environment thus all the external variables are still accessible. In the following example the variable `a` is defined outside the interpreted block. Nevertheless, the value of `a` is properly used during the evaluation of `[a + 1]` in the `Debugger`. Furthermore the following example shows the full debug trace of all the message sends used for the evaluation of `[a + 1]`:

```
> a := 'a'
'a'
> Debugger interpret: [ a + 1 ]
BlockClosure>>#value (0)
pidb>       String>>#+ (1)
pidb>         String>>#doesNotUnderstand: (1)
pidb>           DoesNotUnderstand class>>#new (0)
pidb>             ObjectLayout>>#instantiate: (1)
pidb>               ObjectLayout>>#basicInstantiate: (1)
pidb>                 ObjectLayout>>#initialize: (1)
pidb>             DoesNotUnderstand>>#initialize (0)
pidb>           BlockClosure>>#value: (1)
```

```
pidb>              DoesNotUnderstand>>#message: (1)
pidb>              DoesNotUnderstand>>#receiver: (1)
pidb>              DoesNotUnderstand>>#signal (0)
pidb>                Exception class>>#throw: (1)
pidb>                 Exception class>>#handler (0)
pidb>                  Continue>>#escape: (1)
pidb>
'a' does not understand: #+ (1)
>
```

The abstract principle of a debugger is to run custom actions – in this case a shell – on each message send. We decided to abstract the implementation of handling custom actions on each message send away by creating an abstract `SteppingInterpreter`. Figure 2.2 shows an overview of this implementation. The `Stepping-`

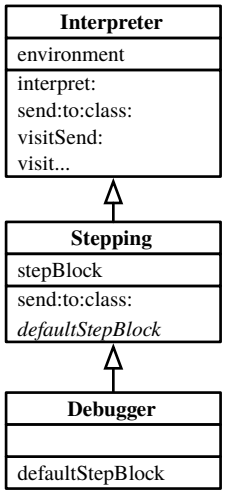

Figure 2.2: UML Diagram of the Debugger related Classes

`Interpreter` overwrites the visitor methods for handling the message send so that custom blocks can be executed.

```
SteppingInterpreter>>send: message
              to: receiver
              class: class
   ↑ self
      checkStep: receiver
      class: class
      message: message
      do: [ super
            send: message
            to: receiver
            class: class ]
```

The method `checkStep:class:message:do:` simply forwards all its arguments to the `stepBlock`. The default implementation of this stepping block simply executes the action passed in, which means that it forwards the message evaluation to the default implementation in the super class.

```
defaultStepBlock
   ↑ [ :receiver :class :aMessage :action | action value ]
```

For the `Debugger` the evaluation of the passed in action is deferred. Instead a shell is launched providing basic functions to inspect the current values and print the stack trace.

```
defaultStepBlock
   ↑ [ :receiver :class :message :action |
      self print: receiver class name, '>>', message.
      self debugShellWithAction: action ].
```

2.4.3 Extending the Debugger

As a case study we implement a *parallel debugger*. Unlike the normal debugger, which only evaluates one block at a time, this special kinds of debugger takes two blocks and interprets them in parallel comparing the state of evaluation at each step.

Consider the following failing test case that we encountered during the development of Pinocchio:

```
bucket := SetBucket new.
bucket at: #key put: 'value'.

self assert: (bucket includes: #key).
self assert: (bucket includes: 'key').
```

The second assertion (last line) fails. This test was documenting a bug that we had difficulties to track down. Symbols and strings are considered equal (`#key = 'key'`) in Smalltalk and hence the second assertion should pass too.

Using the basic debugger described in Section 2.4.2 to find the difference in execution of the two assertions is cumbersome. The manual approach would be to launch a separate debugger for each of the assertions and step through the code until the states of the tests differ.

An effective solution for this specific problem is to create a new specialized debugger, a `ParallelDebugger`. The use of the `ParallelDebugger` for the previously mentioned test case looks as follows:

```
ParallelDebugger interpret:
   (Array
      with: [ bucket includes: #key ]
      with: [ bucket includes: 'key' ])
```

The debugger runs the given blocks in parallel up to the point where the executions start to differ:

```
SetBucket>>#includes:
   SetBucket>>#do:
      SmallInt(1)>>#to:do:
         BlockClosure>>#whileTrue:
            SmallInt(1)>>#<=
              SmallInt(1)>>#>
              --> false
              false>>#not
              --> true
            true>>#ifTrue:
              SetBucket>>#at:
              --> #'key'
              Symbol(#'key')>>#==
              1) (#'key')--> true
              2) ('key') --> false
```

Listing 2.1: Parallel debugger trace

Looking at this trace immediately reveals that both traces differ upon a strict equality check on a symbol. In the first case the comparison returns `true`, in the second case `false`. `SetBucket` incorrectly uses == (pointer equality) rather than = to compare keys, rendering strings and symbols distinct. The `ParallelDebugger` provides the minimal output needed to quickly identify the root cause of the problem.

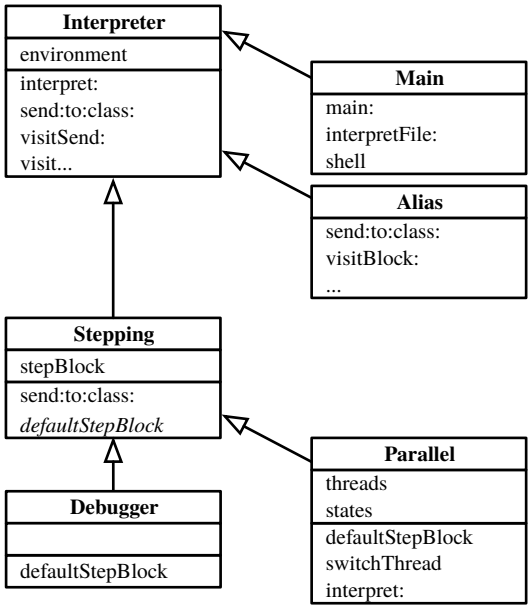

Figure 2.3: A Pinocchio Interpreter hierarchy

Just like the normal debugger (described in Section 2.4.2) the parallel debugger is built as a subclass of the stepping interpreter (see Figure 2.3). The main difference is that the stepping block is not used to control a single execution trace but to handle the interleaved execution of the given number of closures to evaluate. We use the first-class continuations of Pinocchio to run the closures in parallel as threads, switching between the threads at each message send. Whenever we reach the end of the list of threads, we compare the state of all the routines and continue with the first thread. The implementation of the parallel debugger is similar to the implementation of coroutines using continuations [5] but with automatic thread switching before and after each message send.

The code to implement the parallel debugger is shown below. The `interpret:` method initiates the interpretation. It starts by using its `initialize:` method (2) to create a number of threads to interpret the closures to be interpreted in parallel. Next it starts a loop (3) that will lazily start a thread for each closure. This is done by capturing the current continuation inside the loop (4), and using this continuation to initialize the next thread (9). `Continuation on:` captures the current continuation and passes it in as an argument to the block that follows. Here, the current continuation is bound to `startNext`. If there is a thread following the current one (7), then the next thread's continuation and context are initialized. The interpreter then proceeds to interpret the closure in the current thread (13), which will eventually pass control to the next thread, or return, thus terminating all threads.

```
interpret: closures
    self initialize: closures size.
    closures do: [ :aClosure |
        Continuation on: [ :startNext |
            " Pre-install a thread that starts
              evaluating the next closure. (a) "
            threads ifHasNext: [ :thread |
                thread
                    continuation: startNext;
                    context: nil ].
            " Leave the debugger when the first
              thread actually returns. "
            ↑ super interpret: aClosure ]].
```

The parallel debugger uses two instances of `StatefulArray` to manage the debugger's state. A `StatefulArray` is simply an array that maintains a pointer to keep track of the currently selected entry. At each invocation of `next`, the pointer advances. After reaching the end of the list, the pointer returns to the first element. The `threads` array keeps track of the currently executing thread, and the `states` array keeps track of the value computed by the current thread.

```
initialize: size
    threads := StatefulArray new: size.
    states := StatefulArray new: size.
    1 to: size do: [ :index |
        threads at: index put: Thread new ].
```

Just like the normal debugger, the execution of the parallel debugger is handled by the stepping interpreter and governed by `defaultStepBlock`:

```
defaultStepBlock
    ↑ [ :receiver :class :message :action |
        states put: receiver @ class @ message.
        self showInvocation.
        " Switch to compare all invocations. (b) "
        self switchThread.
        states put: action value.
        self showReturn.
        " Switch to compare all return values. (e) "
        self switchThread.
        " Return the result of the invocation. (f) "
        states current ].
```

Whenever the current thread sends a message, the block above is activated. It saves the invocation information into the states array (21), shows some information about the invocation to the user (22), and switches to the next thread (24).

When the parallel debugger returns to the thread, the actual message send is performed and the result of the message send is stored back as the current state of the thread (25). The next thread is then activated again (28).

Finally when the parallel debugger returns to the thread, the result of the method invocation that activated this particular block is returned from the block (30).

As we have seen above, each thread stores the current continuation and its context. Threads are switched by passing control from the current thread to the next thread:

```
switchThread
    | thread |
    Continuation on: [ :aContinuation |
        " Store the current thread."
```

```
        threads current
36          continuation: aContinuation;
            context: context.
38      " Prepare the next thread. "
        states next.
40      thread := threads next.
        " After having stepped through all threads,
42        compare all states to the first. (c) "
43      threads ifAtFirst: [
44          states do: [:each|
45              self assert: (each == states current) ]].
46      " Resume the next thread (d) "
47      context := thread context.
48      thread continuation continue ].
```

First, the current thread captures the current continuation and context and stores them (37). (NB: Note that `context` is an instance variable of the interpreter (see Figure 2.3). Then the arrays containing the states and the threads are advanced to the next position (39). If all threads have advanced one step (43), the states are checked for equality (44). Finally the next thread is resumed by restoring its context and continuing the continuation of that thread (48).

In Figure 2.4 we can see the resulting behavior of the parallel debugger. At point (a) (line 6 in the `interpret:` method of the parallel debugger), the two threads are installed. We then skip ahead to point in the trace (Listing 2.1) where the results of retrieving the keys from the first bucket are compared to the actual key. The invocation `true ifTrue: ...` is displayed and control switches to the second thread at point (b / line 23). The second thread switches back when it reaches the same point (b). Since we now have returned to the first thread, the invocations are compared (c / line 42) to see if they are the same. This means that the receivers (both `true`), the message (`ifTrue:`) and the code of the argument blocks are compared. The invocations match, so the first thread is resumed (d / line 46). The interpreter is now recursively invoked to retrieve the first element in the bucket. Again the invocations match. Now the results of the invocations are computed and saved, control is returned (e / 27), and the results are compared. The results match, so they are returned (f / 29), and the interpreter again invokes itself recursively to check if the retrieved key matches the expected one. The invocations match (since `#'key' = 'key'` in Smalltalk), but the results of the invocations do not match (since `#'key' == 'key'` does not hold).

Evaluation The parallel debugger, like the serial debugger presented in Section 2.4, directly reuses the object model of the underlying base level interpreter. As a consequence, no upping or downing is required, and objects can freely flow between the base and meta levels.

Even though interpreters are defined recursively as AST visitors, this poses no problem for expressing non-local flow of control. Threads are easily simulated by capturing the needed continuations and explicitly transferring control when needed. The parallel debugger is only possible due to the support for continuations in Pinocchio. Without continuations we could not switch between the execution of multiple closures. It would only be possible to continue the execution of the next closure from inside the current one.

2.5 Summary

In the first part of this chapter we discuss the background and certain details of the Smalltalk based high-level language VM called Pinocchio. In Chapter 4 we will use

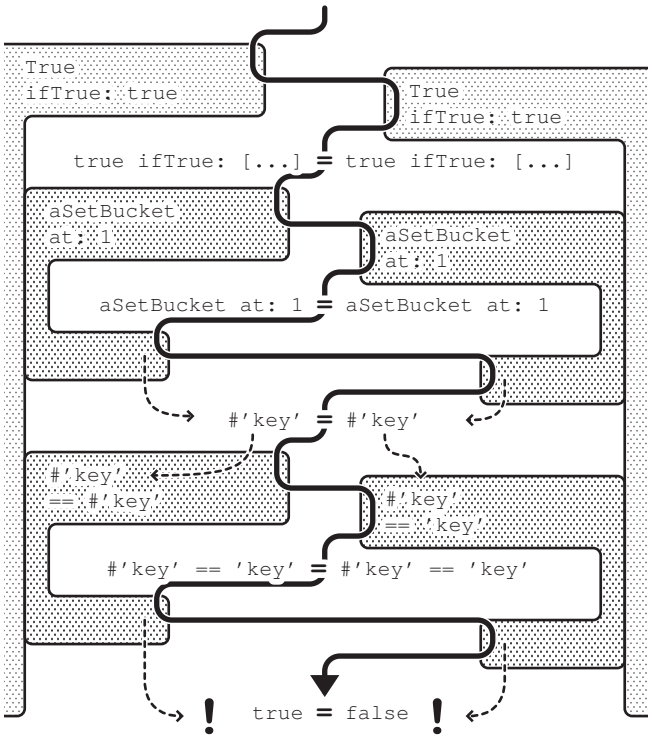

Figure 2.4: Thread-based parallel execution of two code parts in the `Parallel-Debugger`.

Pinocchio to discuss in detail several optimizations. Pinocchio is based on the early Scheme dialect with Smalltalk semantics called SchemeTalk presented in Section 2.2.1. SchemeTalk is not classical VM based language implementation but rather a macro-based extension to a standard Scheme. Since the VM definition language and the interpreted language are conceptually very close objects can flow between both levels without breaking encapsulation. Based on the work of SchemeTalk the first version of Pinocchio was developed. Scheme Pinocchio presented in Section 2.2.2 featured self-evaluating AST nodes which were later complemented with first-class interpreters. Due to the high language-level flexibility with the concept of self-evaluating AST nodes the performance expectations were not met. In the next version of Pinocchio the performance issues and the complex implementation of the predecessor were addressed. In Smalltalk Pinocchio presented in Section 2.2.3 performance has been increased by manually hard coding the evaluation of common AST nodes. To further increase speed the visitor based evaluation approach has been complemented by opcodes. Section 2.2.4 describes how an opcode-based evaluation helps to avoid unnecessary stack operations needed during visitor-based AST evaluation.

The second part of this chapter focuses on the different extension mechanisms of the current Pinocchio implementation. Next to standard Smalltalk MOPs Pinocchio features first-class AST nodes and first-class interpreters. Using first-class interpreters instead of self-evaluating AST nodes improves the performance of code that relies on standard AST nodes.

The shift from the visitor-based evaluation to the opcode-based approach has been a significant performance improvement in the history of Pinocchio. But next to this major change several smaller optimizations have been applied. Later in this thesis in Chapter 4 five optimizations are discussed in more detail. In order to properly understand the effect of these optimizations we have to write and evaluate benchmarks. There are

three key aspects which have to be respected when writing and evaluating benchmarks: reproducibility, statistical sound evaluation and the focus on micro benchmarks. The tools and techniques to respect these aspects are presented in the following chapter.

3 Benchmarking

To properly evaluate the performance of a computer program several measures have to be taken into account. How should a benchmark be run and how should the results be presented? What benchmarks should be chosen to produce meaningful results?

For building a fast VM it is crucial to have a solid unit test and furthermore a solid benchmark coverage. Benchmarks are used to control the performance development of a VM. Generally many programmers tend to use simple scripts with limited applicability to evaluate the benchmark results. Furthermore the bigger part of benchmark results presented in computer science lack proper scientific presentation and are hard to reproduce. Only a fraction of the papers displays the average of all benchmark runs and a confidence interval [3]. Especially the confidence interval is fundamental in order to provide a measurement of the result's qualities. Properly written benchmarks are reproducible and help to locate bottlenecks in a VM implementation. Hence we encourage the use of a benchmarking framework to produce reliable and trustworthy results.

The first subsection explains the statistical background of benchmarks and explains the basic concepts of standard deviation and confidence intervals. In the next section our benchmarking framework `PBenchmark` is presented which follows the mathematical background presented in the previous subsection. The benchmarking framework provides an easy way of writing reproducible benchmarks of application level code much like unit tests. However in order to inspect low-level details of the VM at runtime other tools are required. Hence we present `DTrace` a tool developed to inspect C binaries at runtime with minimal computational overhead.

3.1 Statistic Evaluation

All the benchmarks provided in this thesis are performed using our own benchmarking suite. Similar to unit tests simple benchmark methods are defined and run at least 30 times. Since this is considered to be a high number of probes [3], a Gaussian normal distribution around the average of the results can be assumed. Figure 3.1 shows a visual representation of equation 3.1 with $\bar{x} = 0$ and $\sigma = 1$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\bar{x})^2}{2\sigma^2}} \tag{3.1}$$

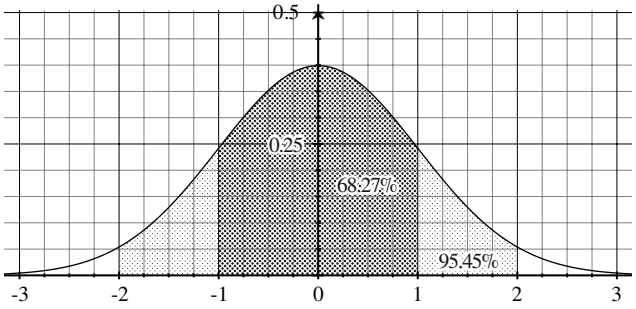

Figure 3.1: Gaussian normal distribution for $\bar{x} = 0$ and $\sigma = 1$

This curve is used to calculate the quality of the benchmark samples. Generally a certain variation around the average of the sample is given together with the percentage of samples lying in this region. Depending on this so-called confidence interval the percentage or significance level changes. In the example in Figure 3.1 the percentage for the interval $[-1, 1]$ is 68.27%. If the confidence interval is increased the percentage of covered samples rises as depicted with the $[-2, 2]$ interval.

In the case of benchmark evaluation it is the ultimate goal to keep the confidence interval small and the significance level as high as possible. This can be either achieved by increasing the number of samples or reducing the variation of the samples. In the second case this means to avoid systematic sampling errors. For instance this can be achieved by using a dedicated benchmarking machine which does not pollute the benchmark results by irregular resource consumption by background processes.

The confidence interval can then easily be calculated from the average $\bar{x}$ and the standard deviation σ and the number of samples n.

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}, \quad \sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}} \tag{3.2}$$

The confidence interval $[c_1, c_2]$ defined by the significance level α is calculated as follows:

$$c_{1,2} = \bar{x} \pm z \cdot \sigma, \quad z = \text{erf}^{-1}(\alpha)\sqrt{2} \tag{3.3}$$

With erf^{-1} being the inverse of the error function. The factor z describes the relation between the standard deviation σ, the confidence interval $[c_1, c_2]$ and the significance level α. Choosing the confidence interval $[\bar{x} - \sigma, \bar{x} + \sigma]$ and thus a $z = 1$ results in a significance level of about 68%. This means that with a 68% probability the next sample is going to be in the provided confidence interval. 3.4 shows the relation between the given values.

$$\alpha = \text{erf}\left(\frac{z}{\sqrt{2}}\right) \tag{3.4}$$

Common values for z and the significance level α are shown in Table 3.1. If not mentioned otherwise, all the benchmarks are run with at least 30 probes and a significance level of 90% is used.

z	$\alpha = \text{erf}(z \cdot 2^{-1/2})$
1.000	68.27%
1.282	80.00%
1.645	90.00%
1.960	95.00%
2.000	95.45%
2.576	99.00%
3.000	99.73%

Table 3.1: Common values for the significance level α and the corresponding factor z of the standard deviation σ

3.2 The PBenchmark Framework

To run the different benchmarks we created a small benchmarking framework in Smalltalk called PBenchmark. It allows you to easily write benchmarks in the same manner as you would write unit tests. The framework follows closely the statistics background mentioned previously in Section 3.1.

3.2.1 Related Benchmarking Frameworks

To compare the performance of two different language implementations it is crucial to provide fair and balanced benchmarks. Generally there is no such thing as a single benchmark fulfilling this requirement. A better way to solve this issue is to define a set of common problems addressed in both languages and compare these results pairwise. This is the approach taken for instance by the PyPy project to check their current performance status. The results of around 25 different benchmarks for Python are run on nightly basis and presented on their website[1]. A similar but somewhat more complete approach is taken by the Computer Language Benchmark Game[2]. A matrix of results for approximately 30 languages, around 10 benchmarks and several OS configurations are presented. The benchmarks mostly solve mathematical problems addressed in game engines. Such a detailed evaluation provides the results to decide upon the speed of a language in a very specific task. Since the benchmarks are implemented slightly differently for each language they can take advantage of specific language features. This is clearly not an objective statement to compare the performance of a language in general. An important lesson learnt from the Computer Language Benchmark Game is that almost every benchmark is flawed. The only way to overcome this limitation is to clearly state what is tested under which exact conditions. Only under these circumstances is it possible to provide reproducible results.

3.2.2 Implementation Details of PBenchmark

The two main classes of the framework are the `PBenchmarkRun` and the `PBenchmarkSuite`. The class `Run` is an extended storage class tracking the execution times/probes of a single benchmark. Furthermore it provides a simple interface to

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

extract the statistical relevant information out of the calculated probes. The other class `Suite` provides an interface to run several benchmarks within a single class. Again this is very similar to a `TestSuite` in `SUnit`. Selectors prefixed with `bench` are considered to be benchmarks. When running a benchmark suite for each selector a `Run` instance is created which then handles the evaluation of this particular benchmark. In `SUnit` this would correspond to a `TestResult` object.

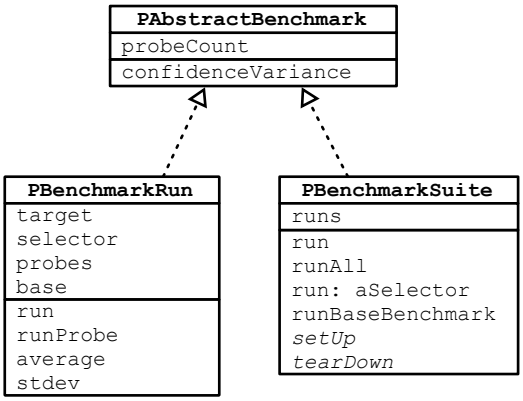

Figure 3.2: UML Diagram for the PBenchmark Framework

The benchmarking framework is heavily used in the dictionary evaluations discussed on page 35. A base benchmark suite is created hosting the different benchmarks. The base suite then is subclassed to specialize the type of dictionary to be used.

While using the PBenchmark framework for preparing the dictionary benchmarks we tried to create a flexible way of using different types of hashes. The most flexible way in this case was to use a special key object where we could manually set the hashes. The previous versions of the benchmark simply used SmallInts as keys which did not pose significant performance overhead compared to the time spent in the dictionary code itself. But with the more complex key objects which are not optimized specially by the VM the benchmark results were no longer comparable to the original ones using SmallInts. A way to solve this issue manually is to calculate the overhead of using the key objects by subtracting the runtime from a benchmark using SmallInts. For an initial setup we made sure that both the SmallInts and the created key objects produce the same hashes. Since both benchmarks would use the same hash values we could assure that the time spent in the dictionary code is the same. Thus by calculating the difference of the two benchmarks the resulting time represents the overhead of the key objects versus the SmallInts. Of course this involves several steps of manual work: creating a comparable base benchmark, running the base benchmarks and the normal benchmarks, calculating the overhead and finally subtracting the overhead from the key object based benchmark. Instead of manually performing these tasks we decided to include this feature directly into the PBenchmark framework. Each benchmark can provide a base benchmark which represents the overhead. In the case of the previous example using key objects a second version of the benchmark is created which creates the same amount of key objects but does not run the dictionary code. This base benchmark then results in the pure overhead introduced by using the key objects. The following code shows the new `run` method with an additional `runBaseBenchmark` message in the end.

```
run
   self reset.
   self runAll.
   self runBaseBenchmark
```

The following excerpt shows in detail how the base benchmark is run.

```
runBaseBenchmark
    | result base |
    (self respondsTo: #baseBenchmark) ifFalse: [ ↑ self ].
    base := self baseBenchmark.
    result := base runAll
    result runs keysAndValuesDo: [ :selector :run|
          (self runs at: selector) base: run.]
```

We first check if the current benchmark implements the `baseBenchmark` method. This is done to stay backwards compatible with benchmarks that do not need a base benchmark. To support a base benchmark the `baseBenchmark` method has to return a valid instance of a benchmark which is then run. Each result of the base benchmark is then attached to the existing benchmark runs. Hence each benchmark run can now provide results in respect to the overhead defined in the related base benchmark run.

An example is given by Figure 3.3 showing the UML-diagram of the dictionary benchmarks. Note that `BaseDictionaryBenchmark` implements the `baseBenchmark` method and all the benchmarks themselves. The `baseBenchmark` method re-

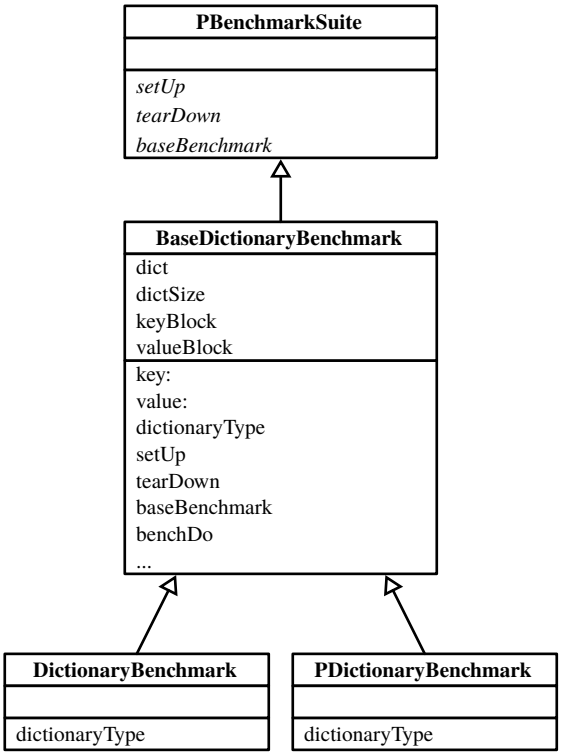

Figure 3.3: UML diagram a use case of the benchmark framework

turns an instance of the `BaseDictionaryBenchmark` and sets some properties on the newly created instance.

```
baseBenchmark
    | inst |
    inst := PBAbstractDictionary new.
    inst keyBlock: self keyBlock.
    inst valueBlock: self valueBlock.
```

In this example the simplest solution is to recycle the base benchmark suite for the normal benchmarks – avoiding a separate implementation of the same benchmarks again. In order to make this work a dummy dictionary had to be implemented. It provides the whole interface used by the benchmark suite but does not provide any functionality. Thus it can be used with the existing benchmarks but does not impose any additional overhead – except for some message sends for using the interface's dummy methods.

3.3 DTrace

The PBenchmark framework presented in the previous section is useful for testing the performance of application level code. When it comes to optimize the low-level details of the VM itself other methods are necessary. A comfortable tool to inspect a program written in C is DTrace. It allows you to probe a C binary with only minimal changes to the sources. DTrace is meant to be used in a production environment as it keeps the probing overhead to a minimum. As long as no probes are activated no considerable overhead can be recorded.

The minimal example of a probe is to track function calls. The following scripts written in the D programming language trace the cumulative and average time spent in each function in the Pinocchio binary.

```
pid$1:pinocchio::entry
{
   self->ts = timestamp;
}

pid$1:pinocchio::return
/self->ts/
{
   elapsed = timestamp - self->ts;
   @tot[probefunc] = sum(elapsed);
   @avg[probefunc] = avg(elapsed);
}

END {
   trunc(@tot, 50);
   trunc(@avg, 50);
   normalize(@tot, 1000);
   normalize(@avg, 1000);

   printf("\nTotal [ms]: "); printa(@tot);
   printf("Average [ms]: "); printa(@avg);
}
```

The `:entry` probe is activated whenever a function is entered. Hence we record a timestamp associated with the called function at that time. Whenever a function in the Pinocchio binary returns, the action under the `:return` probe is evaluated. In this case the `elapsed` time since the function was called is calculated. The next line accumulates the time spent in the current function denoted by `probefunc`. `sum` is not a simple function working on its own but rather belongs to an aggregation defined by the source line `@tot[probefunc] = sum(elapsed);`. When the probing is stopped the action defined by `END` is executed. In this example the aggregation stored in `@tot` and `@avg` is limited by `trunc` to the topmost 50 elements. The next two lines normalize the results to milliseconds as the timestamp used earlier in the code is recorded in microseconds. Eventually the results are printed out.

We used DTrace in several places throughout the performance evaluation of Pinocchio. Most low-level details about the optimizations presented in this thesis were extracted using several different DTrace scripts.

3.4 Summary

In this chapter we have shown how to properly evaluate benchmarks. Starting with the basic mathematical background explaining the core concept of statistical evaluation. Based on this background our Benchmarking framework PBenchmark has been introduced. It allows to write benchmarks very similar to unit tests by writing either single benchmarks or combine them to suites. Relying on a framework helps to reduce evaluation errors and improves reproducibility. We encourage writing benchmarks for very specific use cases in order to help isolating problematic code in the VM implementation. However, it is not possible to track and detect all types of performances flaws in a VM with traditional benchmarking. The can only provide top-down view using the VM implementation as a black box. Hence we further introduced DTrace a low-level binary probing framework. DTrace allows to efficiently inspect the VM internals using customizable probes and a simple query language.

In the next chapter five specific optimizations applied to Pinocchio are discussed using results obtained by the tools and techniques presented in this chapter.

4
Optimizations

After discussing the evolution of Pinocchio and the tools used to benchmark in the previous chapters we focus on the different aspects of performance and optimizations applied in Pinocchio. We focus on the process described in the introduction and the previous section. The benchmark results are obtained by a set of benchmarks implemented using the PBenchmark framework. By using the same set of benchmarks for different types of optimizations we can analyze the performance impact easily. Furthermore we classify the optimizations on the scheme described in Section 1.3 and further specified in the following Section 4.1. With the results presented the efficiency of each benchmark and implementation effort is presented.

Focusing on Pinocchio allows us to discuss many different optimization techniques which were applied throughout the development history. The early version of Pinocchio worked in a very dynamic way by using self-evaluating AST nodes which generate considerable overhead. In the subsequent versions more and more assumptions were directly embedded into C code. Next to avoid dynamic lookups it is important to be able to access often-used object attributes directly in the C level without the need to convert them. Caching data helps to avoid indirect calculations of often used attributes. Caching not only focuses on data allocation but by using inline caches can also focus on execution. Inline caches improving performance by avoiding repeating lookups. The last optimization with significant impact on the performance of Pinocchio is unwrapping values. Hereby pointer indirections on C level are avoided by directly using the low-level value instead of a wrapped version.

To describe these different optimizations a restricted set of three benchmarks is chosen in Section 4.2. In the same section the detailed properties such as the used environment or the number of probes are listed followed by the discussion of the results of the three default benchmarks in Pinocchio and Pharo in Section 4.2.1. The following Section 4.1 lists four important properties of optimizations. For each optimization property an example discussed in the thesis is listed. The next sections cover specific optimizations in more detail. Each optimization is analyzed using the set of the three default benchmarks. Section 4.3 covers the optimizations applied to the dictionary data structure.The following Section 4.1.4 deals with the performance results of caching with SmallInts and characters as a first use case. The next Section 4.7 describes the performance impact of using unwrapped values for the size value of arrayed objects. In the last Section 4.8 future optimization possibilities are listed and their potential performance impact is discussed.

4.1 Type of Optimizations

The VM optimizations presented in this chapter can be separated into two different groups. First there are the transparent optimizations which do not affect the semantics of the interpreted language. Secondly there are the non-transparent optimizations which enforce certain semantic changes. The following two subsections Section 4.1.1 and Section 4.1.2 discuss these two groups in more detail. Next to the effects on the language we can classify optimizations into two other broad groups depending on their implementation details. Optimizations which focus on embedding assumptions presented in Section 4.1.3 and optimizations focusing on caching data presented in Section 4.1.4.

Table 4.1 provides an overview of the types of optimizations presented in this thesis and their corresponding classification.

	Transparent	Non-Transparent	Embedding Assumptions	Caching Data
Native Dictionaries	✓		✓	✓
Opcodes	✓		✓	
Caching Integers	✓			✓
Unwrapping values	✓		✓	✓

Table 4.1: List of the optimizations discussed in this thesis and their classification

4.1.1 Transparent Optimizations

Transparent optimizations are VM-level optimizations which do not require changes in the interpreted language's semantics. An example of such an optimization is presented in Section 4.5 where integers and characters are cached in a transparent manner. Although not all integer objects are treated the same way, there is no possibility for the programmer to see the difference. The most commonly used integers are already preallocated to improve speed. The only way a programmer could potentially distinguish a cached integer from a non-cached version would be to rely on the object identity. However, the == method on `SmallInt` is implemented natively to directly compare the integer value. This is different from the default implementation where the C-level pointer value is used to determine object identity. So even though the optimization would be not fully transparent the small change in the comparison semantics help to guarantee transparency again.

A common optimization which is transparent – not yet present in the current version of Pinocchio– is a just in time compiler (JIT). A JIT alters the evaluation structure at runtime to be more efficient on the underlying hardware. Nevertheless it does not alter the semantics of the language. This is an implicit property of a JIT as the modification of the evaluation structure generally happens at runtime. Hence it would be strange if the semantics of the language would change over time depending on the usage. Thus we can clearly identify that JITs are transparent optimizations.

For a stable language it makes most sense to rely on transparent optimizations. This way the VM can be considered as a black box implementing a predefined set of language requirements. Transparent optimizations help to form a stable language with a clear design. Especially for a high-level system such as Smalltalk it is crucial to separate the high-level language concerns from the low-level VM ones. Otherwise all high-level languages will start resembling a C-like system just because it is easier and faster to build an efficient system for such a language.

4.1.2 Non-transparent Optimizations

In contrast to transparent optimizations the non-transparent ones explicitly require changes in semantics. During the early stages of Pinocchio several non-transparent optimizations were applied to the VM in order to increase performance. Such an example was the remove of self-evaluating AST nodes in Section 2.2.3. As a replacement we introduced first-class interpreters which of course required to rewrite parts of the Smalltalk code.

Generally non-transparent optimizations should be avoided in high-level languages. Depending on the kind of optimization they might pollute the design. The favorable way to apply non-transparent optimizations it by finding a compatible way to reintroduce the lost features. The previously mentioned first-class interpreters in Pinocchio are an example for this.

4.1.3 Embedding Assumptions

High-level dynamic languages try to introduce more flexibility by delaying the point on which to decide what code gets executed. For instance by using polymorphism it is unclear which code gets executed until the current evaluation reaches that point. On the other hand it is possible to directly inline functions in a statically typed language like C. Another issue when using high-level languages is the difference in evaluation compared to the language the VM is written in. In the case of Pinocchio the VM core is written in C. To overcome these semantic differences an intermediate evaluation scheme is chosen, most commonly bytecodes. Bytecodes provide a simple abstraction for high-level operations but still allow for versatile composition. However compared to a pure C level evaluation a significant overhead can be observed. Hence one way to increase performance is to push as much code as possible to the C level. Of course this is in contradiction with the principle of using a high-level language. Furthermore it is not always guaranteed that the pure C-level implementation can be evaluated much faster. Embedding assumptions does not mean to reimplement high-level concepts in the VM definition language but in general to avoid indirections introduced by high-level nature of the implemented language.

Section 4.3 and Section 4.4 provide an overview of two different approaches of embedding assumptions. The first example covers the classical implementation of a high-level data structure directly in C. The dictionary data structures from Pinocchio are available in Smalltalk and in C. The next example covers the use of opcodes. Unlike the first example, focusing on data structures, the focus is on the evaluation scheme itself.

Another type of embedding assumptions is presented in Section 4.7 where the size property is stored using either a high-level or a low-level number representation.

4.1.4 Caching Data

In an object-oriented language such as Smalltalk it is comparably expensive to perform algorithmic calculations. Even though sending messages in Smalltalk is cheap compared to other dynamic languages it is still expensive compared with a C level function call. To avoid subsequent calculations it pays off to cache certain values. An example of that can be seen in the `PDictionary` code presented in Section 4.3. Each bucket knows exactly the number of elements stored in it. When looping over the elements of a bucket it is straight-forward to use the known boundaries to limit the number of traversed items. However, the cached number of elements is not needed for a successful bucket traversal. In a very primitive solution one could figure out the bucket boundaries by comparing each element against `nil`, since the unused position at the end of the bucket are initialized with `nil`. Of course in this case this poses a significant overhead due to the high number of superfluous checks.

Next to optimizing the data structures themselves by caching often used values it pays off to store recycle certain objects as a whole. In a high-level language like Smalltalk integers and characters are first-class objects. This implies a significant overhead as these objects are used with a very high frequency. In Section 4.5 we present how integers and characters are precalculated and reused.

A further version of caching presented in Section 4.6 which does not directly focus on caching values used in data structures. Inline caches help to improve the lookup speed for methods. Instead of performing a certain method lookup over and over again, the resulting method is cached at the send site.

4.2 Default Benchmarks

During the optimization of Pinocchio we cared about the overall performance but for a long time our only use case was the Fibonacci algorithm. The particular choice for this benchmark was rather driven by the simplicity of its implementation than the quality of the results – the very first version of this benchmark was created by tediously assembling the AST in plain C code. There the size of the sources measured in number of AST nodes was crucial. By further optimizing the system it paid off to have very specific micro benchmarks. Thus the outcome of a single optimization can be tracked more easily than with a big benchmark producing only a single result. When using the PBenchmark framework presented previously in Section 3.2 several micro benchmarks can be grouped into a suite. This way it is possible to combine a single results – for basic comparison – with a fine grained view on different aspects of an optimization.

In order to evaluation the different optimizations applied during the development of Pinocchio we focus on a set of three default benchmarks. This way it is possible to compare the performance impact of each optimization. The set of benchmarks used are a parser suite, a dictionary suite and Fibonacci. The complete sources of these benchmarks are listed in the appendix. The parser and the dictionary benchmark consist of around 10 different micro benchmarks. Throughout the evaluation of the optimization the choice of benchmarks has proven to be good enough to display the various impacts on performance. The benchmarks' sources are available from the PBenchmark framework and the Pinocchio project respectively.

4.2.1 Default Benchmark Results

If not otherwise specified all the benchmarks are run on a 64-bit server machine with 8 cores à 2.3 GHz and 16 GB of installed RAM whereof around 800 MB are consumed by the running system. The running operating system is an `Ubuntu 10.04.1 LTS` with the `2.6.32-24-server` Linux kernel. At most four benchmark evaluations are performed at the same time keeping the overall load inclusive background tasks maximally at 60%. Generally the benchmarks are run 1000 times taking the average and displaying the variation for a significance level of 90%.

Table 4.2 and Table 4.3 show the results of the three different benchmarks for Pinocchio and Pharo respectively. The tables show the absolute time used for each of the benchmark suites. The value in parentheses shows the variation in the last two digits for the 90% significance level. Figures like Figure 4.1 present the relative time compared to the results given by the default run times shown in Table 4.2. For instance compared to Pinocchio, Pharo runs around 100% or 2 times slower in the Dictionary benchmark using `PDictionaries`.

The results for Pharo presented in Table 4.3 show that Pinocchio is around 3 to 6 times slower than Pharo in tasks like message sends and integer addition – the most common actions in the Parser and Fibonacci benchmarks. In the dictionary benchmarks Pharo is 2 to 4 times slower depending on the choice of dictionaries. This is due

to native implementations of part of the dictionary code in Pinocchio. The results presented in Section 4.3 should provide a more thorough presentation of the different dictionary implementations on the two platforms.

Benchmark	Absolute Time
Parser	2.9184(15)s
Dictionary	0.1707(17)s
Fib 31	0.82383(79)s

Table 4.2: Benchmark evaluation with all optimizations enabled in Pinocchio

Benchmark	Absolute Time	Ratio compared to Pinocchio
Parser	0.46317(13)s	0.159
Smalltalk - Dictionary	0.3262(42)s	1.911
Dictionary	0.62680(41)s	3.671
Fib 31	0.32047(11)s	0.389

Table 4.3: Benchmark evaluation from within Pharo-1.1 in comparison with Table 4.2

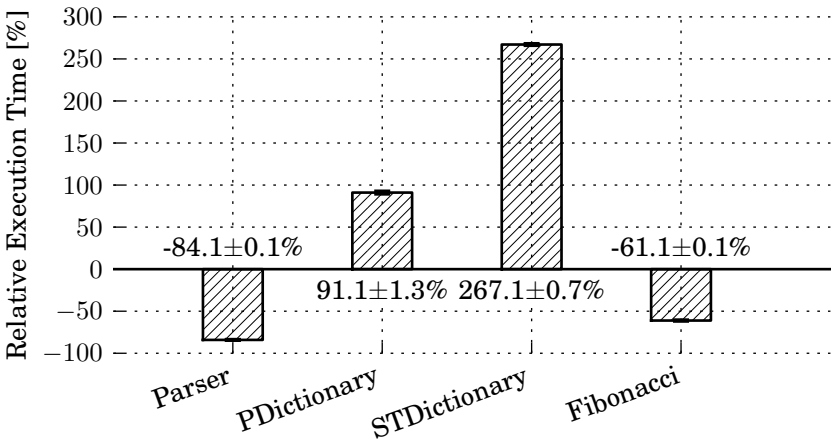

Figure 4.1: Benchmark evaluation from within Pharo-1.1 compared to Pinocchio

4.3 Native Dictionaries

To increase performance in a dynamic language it is crucial to implement as much as possible in C code. However this might harm the dynamic nature of a language if not done carefully. In Pinocchio much work has gone into implementing efficient dictionaries and sets in C code. Dictionaries are essential components of the language – used as method dictionaries the access speed highly affects the overall speed of the VM, although there are ways to avoid repeating method lookups using inline caches as discussed in Section 4.6. Another place where such a data structure affects performance – although considerably less – is the symbol set. For each conversion from a string to symbol a lookup in the symbol dictionary is performed. In both cases it makes sense a highly optimized version directly in C since only a limited number of types for the keys

are used to access the items in the dictionary. For both method and symbol dictionaries the keys or elements are Symbols. The C version natively supports Symbols, Strings and SmallInts as keys. Specialized C functions can directly extract the hashes from these objects without leaving C. In the case there is an "unknown" type of object the default MOP needs to be activated. Hence `hash` is sent to the key objects of types other than Symbol, String or SmallInt. Of course this involves some overhead since the C level has to be left and the hash value is retrieved asynchronously. Thus the C code has to be written in a stack ripped style. At each message send back to the application level the C function needs to be split up into subfunctions. This is the only way to let the VM fully handle control flow interleaved with native C code.

4.3.1 Implementation

The dictionary implemented in Pinocchio is bucket-based and uses a hybrid scheme between a linear collection and a dictionary lookup. For a certain number of element (the default is 40) the dictionary behaves like a `SmallDict`: Instead of using several buckets only a single bucket is used. This pays off for as long as the linear search over a bucket is faster than retrieving the hash and calculating the proper basket.

Both, sets and dictionaries internally work with an `Array` of buckets. The number of buckets is always a power of two, making it possible to directly index into the bucket by masking the hash value of the key. In the dictionary bucket itself keys and values are stored sequentially in an alternating fashion. If the buckets start to degenerate by having too many elements, the total number of buckets is doubled and the elements are redistributed. During this rehashing only the elements with a changed bucket index need to be moved. Since the total number of buckets has been doubled the new position of the key-value pair depends only on the newly masked bit of the hash.

Currently the native code supports normal dictionaries, sets and identity dictionaries. Although most of the functionality has been implemented in Smalltalk code a subset of the methods for putting and retrieving elements is implemented directly in C code. Explicit type checks for SmallInts, Strings and Symbols are performed to extract the precalculated hashes without indirections over the application level. Since dictionaries are already used during the bootstrap for the symbol table and method dictionaries, part of the functionality is implemented as pure C functions:

```
1  void IdentityDictionary_store(IdentityDictionary self,
2                          Optr key, Optr value)
   {
4     assert0(self != (IdentityDictionary)nil);
      long hash = get_identity_hash(self, key);
6     DictBucket * bucketp = get_bucketp(self, hash);
      if (*bucketp == (DictBucket)nil) {
8        *bucketp = new_bucket();
         DictBucket bucket = *bucketp;
10       bucket->values[0] = key;
         bucket->values[1] = value;
12       bucket->tally = 2;
         return IdentityDictionary_check_grow(self);
14    }
      if (Bucket_quick_store(bucketp, key, value)) {
16       IdentityDictionary_check_grow(self);
      }
18 }
```

`IdentityDictionary_check_grow` can be directly called from within C code. The only computed values used during this operation are the objects' identity hashes which can be calculated without the need of application level code. For the general

usage during program execution a second version of the aforementioned function is implemented as an opcode:

```
OPCODE(dictionary_store)
   Optr w_hash = PEEK_EXP(0);
   Dictionary self = (Dictionary)PEEK_EXP(4);
   long hash = unwrap_hash(self, w_hash);
   Optr key = PEEK_EXP(3);
   Optr value = PEEK_EXP(2);

   DictBucket * bucketp = get_bucketp(self, hash);

   if (*bucketp == nil || (*bucketp)->tally == 0) {
      add_to_bucket(bucketp, key, value);
      ZAPN_EXP(3);
      POKE_EXP(0, value);
      pc += 2;
      return;
   }

   pc += 1;
   POKE_EXP(0, 0);
   POKE_EXP(1, bucketp);
   Bucket_compare_key(key, (*bucketp)->values[0]);
END_OPCODE
```

This of course only implements parts of the whole functionality. Since the C stack cannot be used in the general case the code has to be split up into small asynchronously executed bodies. Looking at the `Dictionary_at_ifAbsent_` native reveals its indirect nature:

```
NNATIVE(Dictionary_at_ifAbsent_, 5,
   t_push_hash,
   t_dictionary_lookup,
   t_bucket_lookup,
   t_pop_return,
   t_dictionary_ifAbsent_)
```

The highlighted `t_push_hash` opcode has to be executed separately and cannot be inlined directly. Extracting the hash in C can happen only for certain known types of keys but for all the other cases a `hash` message has to be sent back to the key object on the application level. This makes it necessary to have several subfunctions for originally a single C function. The next code excerpt shows the implementation of the `push_hash` opcode.

```
OPCODE(push_hash)
   pc += 1;
   SmallInt hash;
   Optr key = PEEK_EXP(0);
   Optr tag = GETTAG(key);
   if (TAG_IS_LAYOUT(tag, Words)) {
      hash = String_hash((String)key);
   } else if (TAG_IS_LAYOUT(tag, Int)) {
      hash = (SmallInt)key;
   } else {
      Class_direct_dispatch(key, HEADER(key),
                            SMB_hash, 0);
      return;
```

```
14      }
        PUSH_EXP(hash);
16  END_OPCODE
```

After extracting the key from the stack its layout type is checked for either belonging to a String/Symbol or for a SmallInt. In both these cases the VM knows how to directly extract the hash value from the object. In the other cases the `hash` message is sent to the key object which will then push the hash as a result on the stack. By introducing these type checks we basically perform manual JITting and thus avoid indirections. The code presented here is equipped with guards for specific types which then can execute directly C code. Not only does the control flow stay longer on the C level but also can the C compiler give another improvement since bigger functions much better optimized.

4.3.2 General Evaluation

Comparing the impact of natives on the speed of the dictionary implementation in Pinocchio the three default benchmarks are run without any natives. Figure 4.2 shows how much each benchmark suite uses dictionaries. The corresponding numbers are presented in detail in Table 4.4. Fibonacci runs at almost the same speed independent of the performance of the dictionaries. This clearly is in sync with the simplicity of this benchmark. The algorithmic code used in the parser takes minimal advantage in dictionary code. Dictionaries are only used in a limited way for caching. The overall calls to dictionary code must still be very small as there is no measurable performance decrease with natives deactivated. The third benchmark suite focusing on dictionaries exclusively, of course is heavily affected by the disabled optimization. We can see that the overall impact is around the factor ten in the most optimal case. Since the keys used for the dictionary benchmark suite are integers the natives of Pinocchio do not have to leave the VM for most dictionary operations.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	`2.9122(15)s`	`2.9184(15)s`	`0.998`
Dictionary	`1.9380(22)s`	`0.1707(17)s`	`11.351`
Fib 31	`0.81941(91)s`	`0.82383(79)s`	`0.995`

Table 4.4: Benchmark evaluation without natives for Dictionaries

4.3.3 Detailed Comparison of the Dictionary Implementations

The results from the general evaluation show that the Pinocchio implementation of dictionaries is around 10 times faster than the Pharo versions in the optimal case. But the dictionary benchmark used in the previous section is heavily flawed if it should represent an overall use case. By using SmallInts as keys we overly take advantage of Pinocchio's natives optimized specifically for this case. In Pharo there is no specially optimized code for this situation. To create a meaningful comparison several micro benchmarks are presented in more detail in the following text. Focusing on micro benchmarks is particularly useful if the goal is to increase performance and not to simply compare two different implementations with a single, simplified number. It is important that each benchmark does not provide an overall real-world application usage but rather a specific use case with well-defined properties. This helps to define the strengths and weaknesses of the different dictionary implementations in Pinocchio and Pharo. The presented micro benchmarks are:

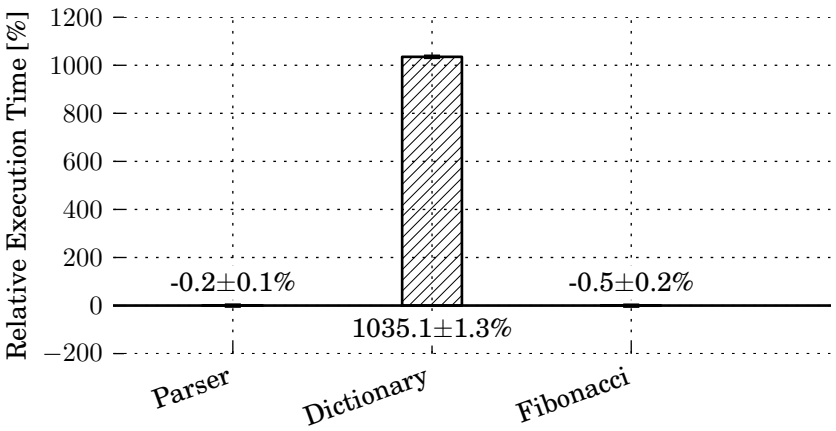

Figure 4.2: Performance impact with disabled natives for dictionaries

1. `at:put:` stores a value with the given key. Half of the keys are already occupied half of the keys are new.

```
1 to: self dictSize * 2 do: [ :i|
   dict at: (self key: i)
      ifAbsentPut: (self value: i)].
```

2. `includesKey:` tests if a key is included using 50% new keys

```
1 to: dict size * 2 do: [ :i|
   dict at: (self key: i)
      ifAbsentPut: (self value: i)].
```

3. `do:` iterates over all elements

```
4 timesRepeat: [
   dict do: [ :i| ]].
```

4. `includes:` tests if an element is included using 50% unknown elements

```
1 to: self dictSize * 2 by: 73 do: [ :i|
   dict at: (self key: i)
      ifAbsentPut: (self value: i)].
```

Each benchmark is presented with a different flavor of keys. Unlike in the default dictionary benchmark we do not use SmallInts directly but rather a key object with customizable hash values. This way the number of unique hashes can be controlled in all detail. The algorithms to produce the hashes are the following:

`(i << 18) + i`	producing no hash collisions
`(i % 100) << 18) + i`	producing only 100 distinct hashes leading to significant hash collisions
`(i % 10) << 18) + i`	producing only 10 distinct hashes leading to a very high collision rate

The micro benchmarks are run with our version of the dictionary marked with `PDict` and the default Pharo dictionary marked with `STDict`. Due to the nature of implementation only the `PDict` is run in Pinocchio whereas both dictionary implementations are evaluated in Pharo. Since `SmallDictionary` performs well only for very small

number of elements and degrades very quickly for larger numbers it has been excluded from the following benchmarks. The benchmarks are presented with an exponentially increasing number of elements ranging from 10^1 to 10^3. Figure 4.3 shows an example

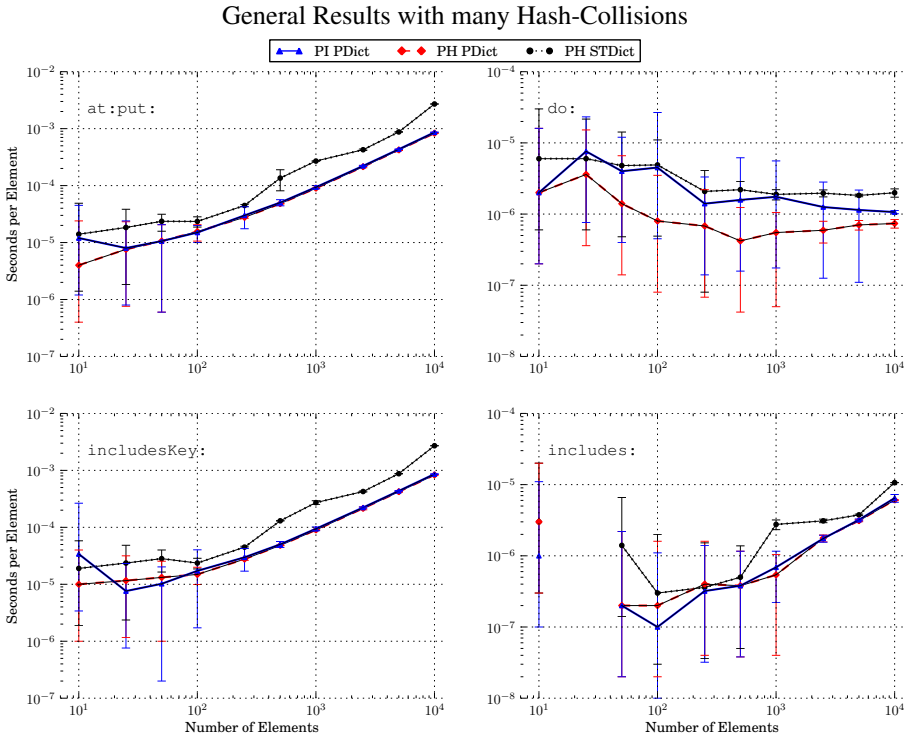

Figure 4.3: Example performance evaluation of all three flavors of dictionaries with the four different micro benchmarks. To generate interesting results only 10 different hashes were used to force quick degradation.

presenting a complete benchmark run displaying all three flavors of dictionaries and all four micro benchmarks. To create an interesting result a very inefficient set of only ten distinct hashes is used in this benchmark. On this figure we can observe that the four micro benchmarks can be split into two groups. On the left are the `at:put:` and the `includesKey:` both showing a similar increasing execution time for each element as the overall number of elements increases. On the other side are two benchmarks which involve looping over all the items in a dictionary.

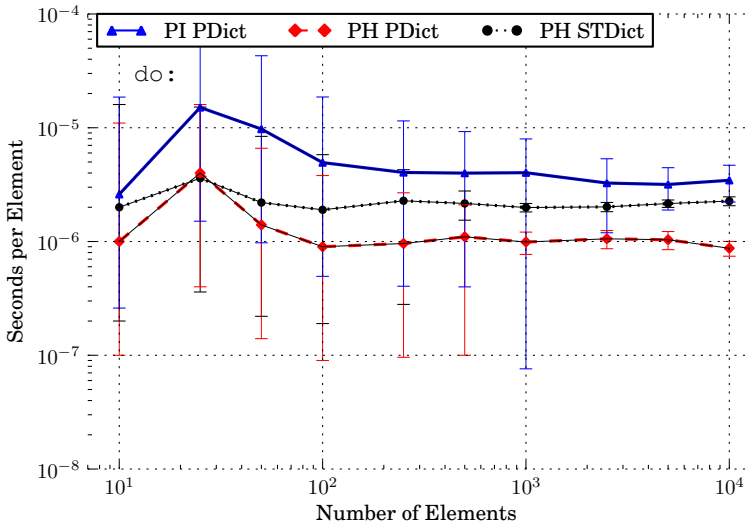

Figure 4.4: Dictionary Performance: evaluating `do:` without hash collisions

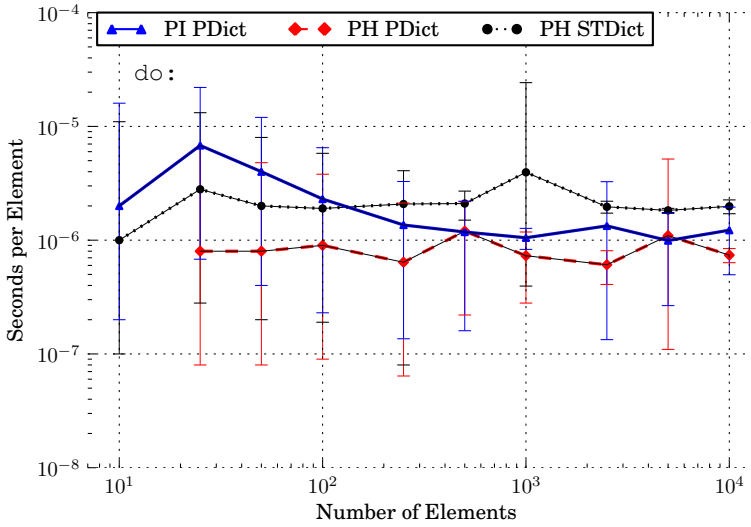

Figure 4.5: Dictionary Performance: evaluating `do:` with heavy hash collisions

Figure 4.4 and Figure 4.6 show in detail the two looping micro benchmarks for key objects with randomly distributed unique hashes. The `do:` benchmark shows a linear behavior since the time spent per element is constant. Due to the high error, resulting from the short runtime of the benchmark only the `PDict` in Pharo serves as a proper performance indicator. In the other cases the deviation depicted by the vertical bars is too high. We can see that the time per element is constant or even decreases with more elements to loop over. With perfect hashes the Smalltalk implementation is faster than the `PDict`. Figure 4.5 shows the results with heavy hash collisions resulting from using key objects with only ten different hashes. In this case the performance of the Pharo dictionaries stays the same whereas the `PDict` performance slightly improves. The performance increase of `PDict` can only be explained by assuming that the looping over the buckets involves some significant overhead. Since there are only 10 filled buckets in the latter example the overhead of looping over the buckets diminishes. Whereas with perfect hashes many buckets are used. Furthermore the version

run in Pinocchio cannot benefit from the power of natives as the `do:` operation invokes a block with each item as an argument. This cannot be handled directly in the native code.

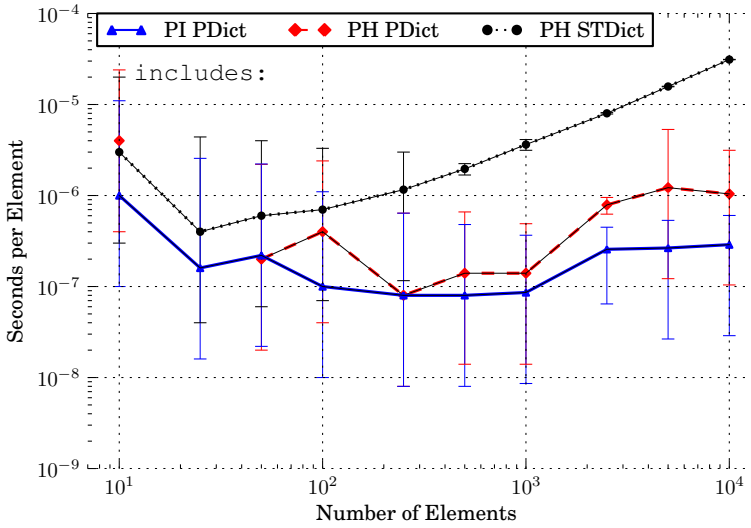

Figure 4.6: Dictionary Performance: evaluating `includes:` without hash collisions

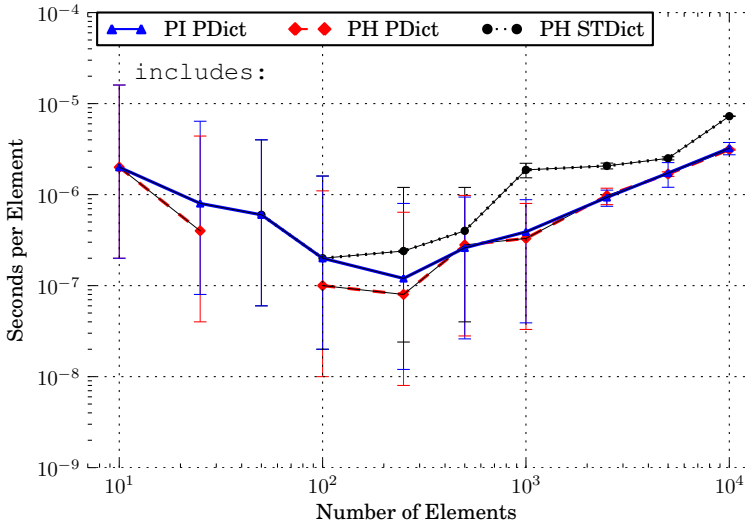

Figure 4.7: Dictionary Performance: evaluating `includes:` with heavy hash collisions

Figure 4.6 and Figure 4.7 show the results for the same setup but this time for the `includes:` benchmark. The straightforward implementation of `includes:` uses the `do:` method to loop over all the elements and returns `true` when the item is found. Although both the `includes:` and the `do:` benchmark are highly similar, they do not show exactly the same results. In general the `includes:` benchmark is slightly slower due to the additional block invocation and comparison overhead. Since these two benchmarks focus more on the execution speed of the VM rather than the data structure we will focus only on the second group of benchmarks, the `at:put:` and the `includesKey:`. In this case we use the dictionary in the most optimal way

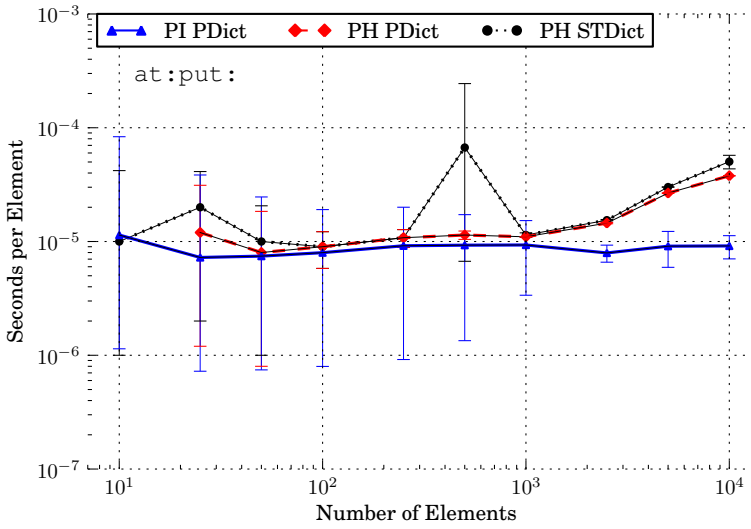

Figure 4.8: Dictionary Performance: evaluating `at:put:` without hash collisions

by accessing stored elements using a key. In the optimal setup using `PDicts` this happens in constant time. The key is directly used to index into the correct bucket where a linear search over the bucket's items is performed. Thus in the optimal case the lookup time is defined by these two factors. The indexing into the proper bucket is a linear operation and so is the linear search. The dictionary is rehashed – the items are distributed over more buckets – when there are too many elements stored. Thus there is a limited number of elements per bucket in the optimal case which results in an upper boundary for the linear search.

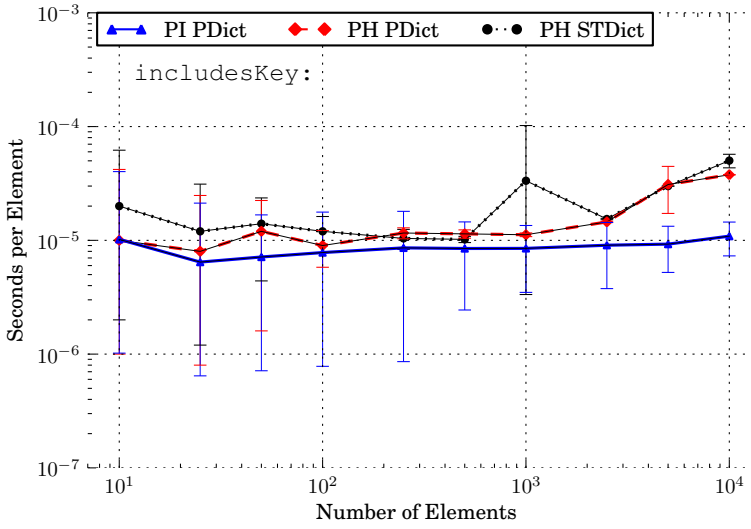

Figure 4.9: Dictionary Performance: evaluating `includesKey:` without hash collisions

Figure 4.8 and Figure 4.9 show the results for optimal hashes. Both graphs are almost congruent hence we will only focus on the `at:put:` benchmark. Even though the benchmark is already generalized – half of the added hash values are already present in the dictionary – it is fully sufficient to show how the dictionary implementations react

to different quality of hashes.

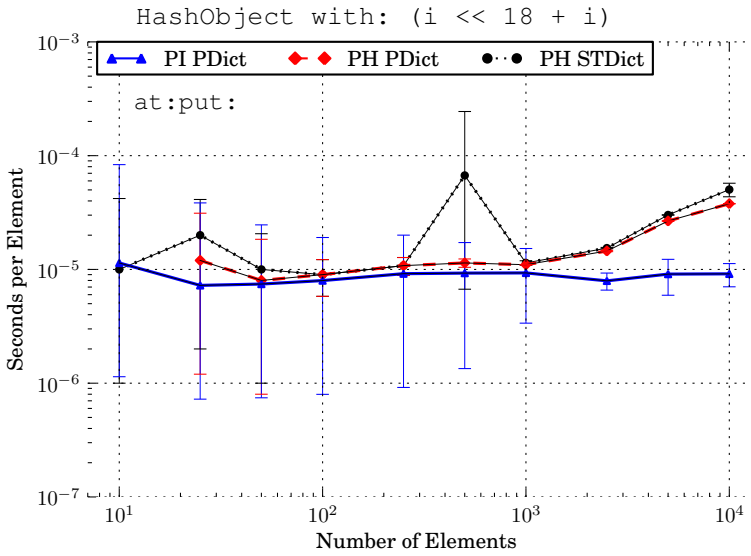

Figure 4.10: `at:put:` dictionary benchmark using perfect hashes.

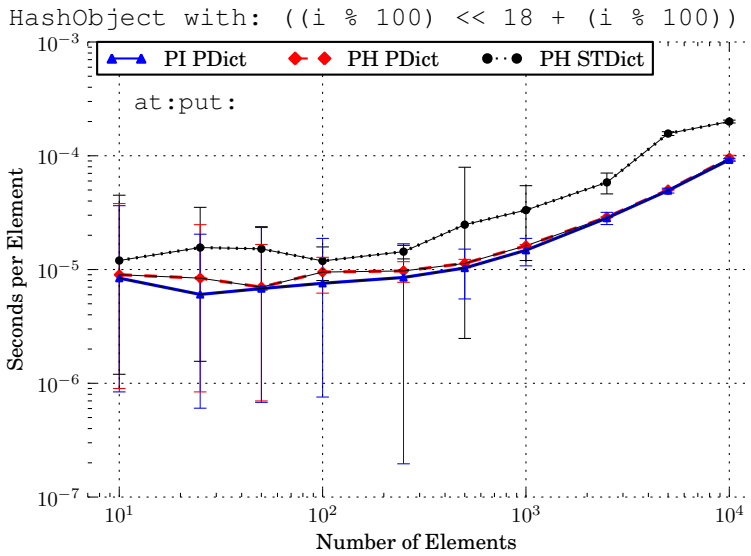

Figure 4.11: `at:put:` dictionary benchmarks using only 100 distinct resulting in moderate hash collisions.

Figure 4.10, Figure 4.11 and Figure 4.12 show the performance degradation of the dictionary implementation with decreasing hash qualities.

In the first graph on Figure 4.10 perfect hashes are used. The hash generating code is shown in the title of the graph. Except for one outlier in the Smalltalk line, the results are very consistent. The Pinocchio `PDict` line shows almost the expected behavior. The time to look up a single item in the dictionary stays constant over an increasing number of element stored in the dictionary. This is also the case for the versions run on Pharo when there are only a limited number of elements used. Starting from 1000 elements the dictionaries seem to degrade a bit. Even though there is a similar movement visible in the Pinocchio line, the errors do not allow for proper comparison. Since the

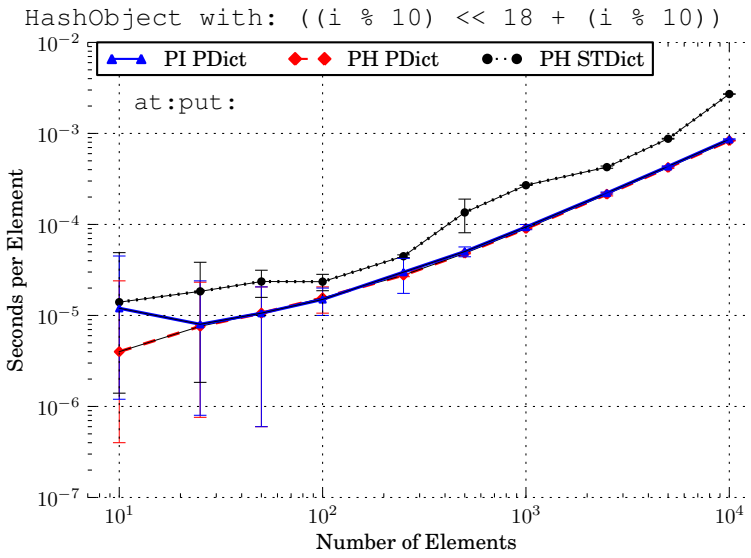

Figure 4.12: `at:put:` dictionary benchmarks using only 10 distinct hashes resulting in a high collision chance.

dictionary degradation is almost the same in Pharo for both dictionary implementations we assume a common source. Most probably the same effect is hidden in Pinocchio since the dictionary is accessed using native code.

In the second graph on Figure 4.11 only 100 distinct hashes are used and thus the number of hash collisions is increased. Comparing this graph to the previous one on Figure 4.10 clearly shows a different performance. This time both `PDict` versions have a almost congruent performance line. The dictionaries start to degenerate after around 250 elements. The final curve shows that the access time is linear to the amount of elements in the dictionary. As soon as the `PDict` degenerates the linear search part of the lookup is no longer a constant but the dominant linear factor. Even though the Smalltalk dictionary does not use buckets it manages to show the same performance response, although in average with a 3 times slower lookup time compared to the `PDict`.

In the last graph on Figure 4.12 the hash collision rate is increased by only using 10 different hashes. The performance response is very similar to the previous version presented in Figure 4.11. This time the degeneration is already visible at less than 100 stored elements. The curve is a bit steeper than the one from the previous version. In Figure 4.11 most probably the curve would approximate the one from Figure 4.12 if a few more data points would be added. Again the Smalltalk dictionary performs around 3 times slower than the `PDict`.

It is clearly visible that the hash quality has a direct impact on the speed of key lookups. The initial constant lookup time in the `at:put:` benchmark degrades to a linear overhead as the buckets sizes degrade. From the graphs presented so far `PDict` performs around 3 times better in lookup related operations. But even in the other two operations the Smalltalk implementation performs slightly worse as shown by Figure 4.3. So far the keys are generated using a list of consecutive integers. For the `PDict` the order of hashes has no direct impact on the performance, only the hash quality does. However since the Smalltalk implementation does not rely on buckets the order of the hashes has an impact of the performance. To avoid a premature judgement a second set of benchmarks is performed using randomized hashes. Figure 4.13 shows the graph for the same benchmark as Figure 4.11 but this time with randomized input. Both versions of the `PDict` perform almost the same. The Smalltalk dictionary performs not better than with consecutive hashes. Thus we can conclude with a high probability that the

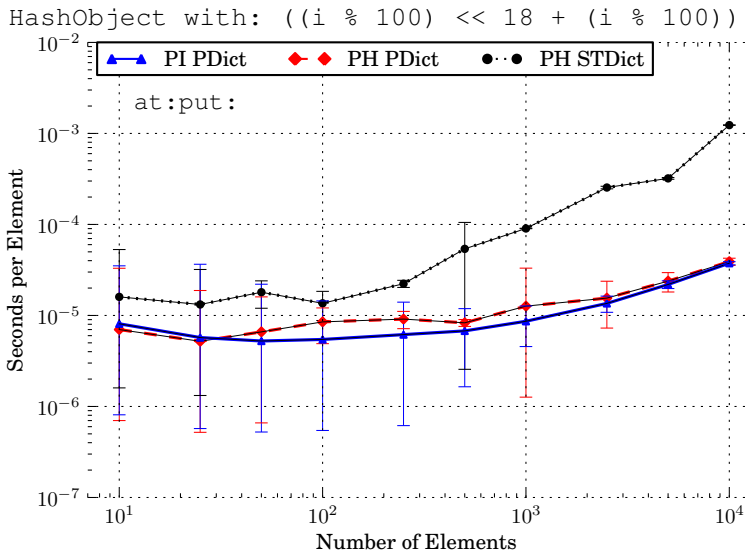

Figure 4.13: `at:put:` dictionary benchmark using randomized hashes with moderate hash collisions.

bucket based approach of the Pinocchio dictionaries is faster than the association-based Smalltalk implementation.

For further comparison of the performance degradation the graphs on Figure 4.14 and Figure 4.15 show the performance response for all four micro benchmarks with the decreasing hash quality. This time the benchmark results are grouped by dictionary flavor rather than benchmark type.

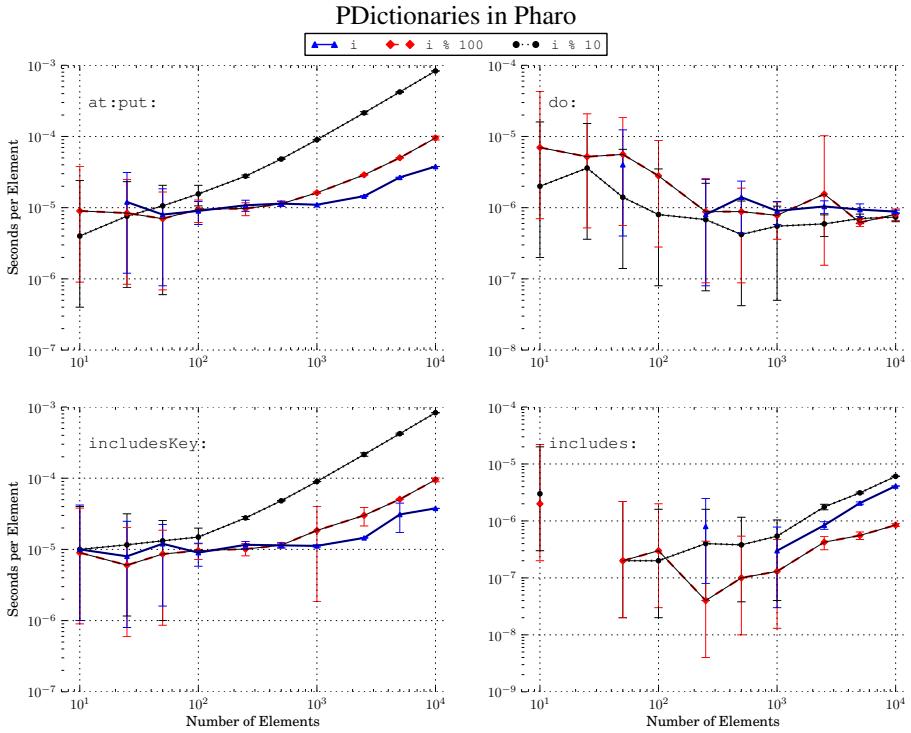

Figure 4.14: Performance degradation with decreasing hash quality on PDictionaries run in Pharo.

4.3.4 Issues

As shown by the ideal benchmark for dictionaries we can achieve a speedup of a factor of ten. Natives are most effective when there is no need to leave the VM level code for application level message sends. Hard-coding known behavior for certain types can greatly improve performance, however, might impose a slight overhead and thus performing worse in other cases. In the case of dictionaries extracting the hash for SmallInts, Symbols and Strings – the most common keys – is greatly accelerated by hard-coded segments. The number of these specialized functions should be kept minimal as well as the number of different hard-coded types. Generally they lead to a rather hard to maintain dual system. Regarding dictionaries it occurred several times that high-level optimizations in the Smalltalk code were not ported down to the C level – just because stack ripped programming tends to be tedious. It would greatly improve maintainability if the C code could be written in a sequential style, directly with inlined application-level message sends. These message sends indicate the points where the control flow is handed back to the application level. Hence this implies that the C-level control flow is interrupted and only resumed after the application-level message send has returned. The following example shows an example of a native require the hash of an argument.

```
1 NATIVE(source_stackripped,
2     Optr arg = NATIVE_ARG(0);
      ...
4     // application level message send
      Optr hash = SEND(arg, SMB_hash, 0);
6     ...
  )
```

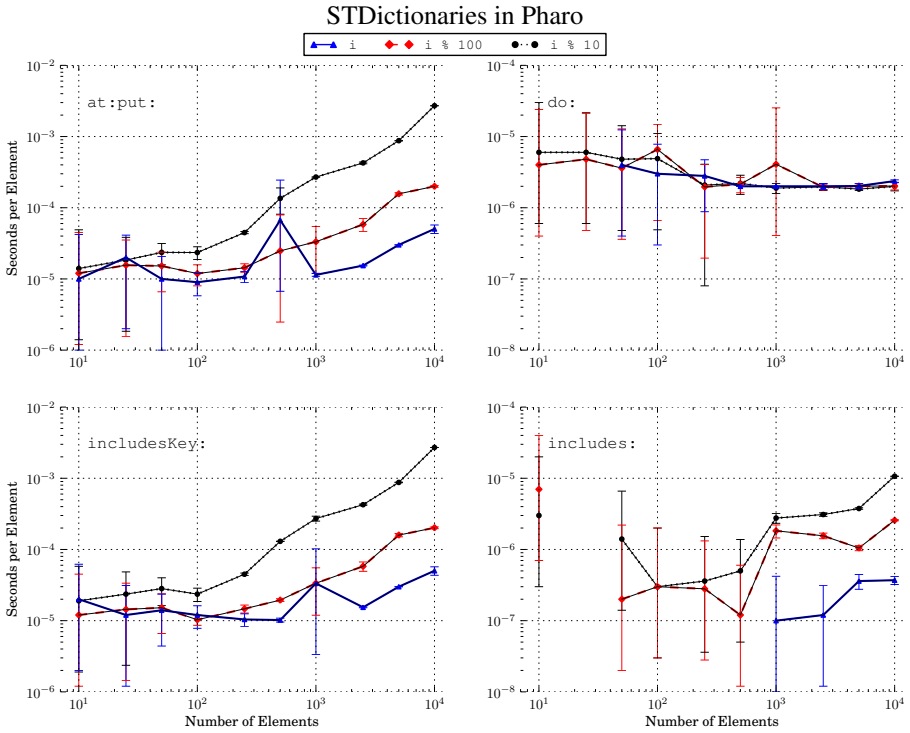

Figure 4.15: Performance degradation with decreasing hash quality on STDictionaries run in Pharo.

The resulting code would then be a stack-ripped version consisting of split-up parts of the function to handle the indirections.

```
OPCODE(stackripped_1,
    Optr arg = NATIVE_ARG(0);
    ...
    // Perform an application level message send
    Class_direct_dispatch(arg, HEADER(arg), SMB_hash, 0);
);

OPCODE(stackripped_2,
    // retrieve the hash from the previous message send
    Optr hash = POP_EXP(0);
    ...
);

NNATIVE(source_stackripped, 2,
    t_stackripped_1, t_stackripped_2);
```

Each of the sub functions ends in an application level call, in the previous example denoted by the `Class_direct_dispatch`. Furthermore it is visible that the C-level control flow is interrupted by the application-level message sends and only later resumed in a second function body. Even this small example results in rather unreadable sources with many unnecessary indirections. If this transformation could be done automatically only the first version would be needed, which helps to keep the sized of the C code small and understandable.

As mentioned in Section 2.2.2 the Scheme Pinocchio featured a style of writing natives in a mixed style with inlined high-level statements. This would be the first step

towards stack-ripped styled code. Next to expanding the high-level statements the C function would have to be split up and probably push needed arguments onto the stack.

4.4 Opcodes

As mentioned in the previous optimization presented Section 4.3 Pinocchio uses opcodes. Opcodes helped to greatly improve performance of the original AST visitor. Similar to bytecodes the control flow is linearized and thus allowing for more optimizations. The choice of opcodes over bytecodes was driven by the possible faster dispatch speed of a direct threaded system.. When using bytecodes or so called dispatch-threaded evaluation, the appropriate behavior for each bytecode has to be decoded. Generally this is achieved by a verbose switch in the main dispatch loop of an interpreter.

```
// evaluatable code
int method_body[] = { 1, 2, 30, 1, ...};

void evaluate() {
   while(true) {
      unsint bytecode = fetch_next();
      switch(bytecode) {
      case 0:
         behavior_0();
         break;
      case 1:
         behavior_1();
         break;
      ...
      default:
         trigger_invalid_bytecode();
      }
   }
}
```

In the most primitive way the incoming bytecodes are compared with several values until the proper switch body is found. The use of a dispatch table is encouraged in this case. Rather than testing until the right function is found the bytecodes are used to directly index into an array of function pointers as illustrated by the following example.

```
// bytecode function definitions
void behavior_0() {
   ...
   eval_next_opcode();
}
...

// lookup table linking bytecodes with functions
lookup_table[0] = &behavior_0;
lookup_table[1] = &behavior_1;
...

// evaluatable code
int method_body[] = { 1, 2, 30, 1 ...};

void evaluate() {
   while(true) {
```

```
        unsint bytecode = fetch_next();
19      if (bytecode > lookup_table_length {
           trigger_invalid_bytecode();
21      }
        lookup_table[bytecode]();
23   }
}
```

Of course a intelligent enough compiler should be able to automatically transform the dispatch switch from above into a dispatch table. Depending on the optimizations performed by the C compiler a dispatch switch with inlined C functions might be faster than a dispatch table. The next step to optimize is to get rid of the lookup indirection at all. Instead of using an intermediate code as binary format which has to be translated to the associated behavior, the function pointers themselves can be used directly. Thus the bytecodes can be replaced by function pointers. In the following example the previously inlined switch bodies have been split up into separate small functions.

```
1  // opcode definitions
2  void behavior_0() {
      ...
4     eval_next_opcode();
   }
6
   void opcode_1(){
8     ...
      eval_next_opcode();
10 }

12 // evaluatable code
   void *method_body[] = {
14       &behavior_0, &behavior_1, ...
      };
16
   void eval_next_opcode() {
18    pc++;
      method_body[pc]();
20 }

22 void evaluate() {
      method_body[0]();
24 }
```

The evaluation strategy is almost the same as with bytecodes. The next instruction is fetched and evaluated. But in this case there is no need for a second indirection to translate the numeric value of the bytecodes into an executable function. To further optimize the opcode evaluation we can use direct threading. Hereby the dispatch is optimized fully away by letting each opcode directly jump to the next instruction preferably using goto statements. [9] shows that direct threaded evaluation can be up to two times faster than a bytecode dispatch based evaluation. The following code shows a variant of the previous examples in a direct-threaded fashion. Now again the behavior bodies are included in a single function. Instead of activating a full function the `NEXT` statement directly fetches the next goto target and jumps there.

```
1  void *method_body[] = {
2        &&behavior_0, &&behavior_1, ...
      };
4
```

```
// directly evaluation the next opcode using gotos
#define NEXT goto **++pc;

void evaluate() {
   void **pc = method_body - 1;
   NEXT;

   // opcodes using goto labels
   behavior_1:
      ...
      NEXT;
   behavior_2:
      ...
      NEXT;
   ...
}
```

The general issue with the faster dispatch is that the representation of the code is no longer platform independent as it previously was with bytecodes. However it is possible to translate bytecodes into opcodes suitable for direct-threaded evaluation. Of course this can happen lazily at runtime keeping the portable nature of bytecodes and reducing a possible startup delay. However using bytecodes as an intermediate format might not be necessary at all. Similar to JavaScript VMs in popular browsers a fast enough compiler can directly output opcodes from the source code.

The compiler used in the version of Pinocchio presented here was written in around one day and can thus be considered as rather primitive and immature. Nevertheless we could achieve a remarkable speed improvement by adding simple optimized special opcodes. The choice for the following optimization was driven by the choice originally made for Pharo VM where the same special bytecodes exist.

```
visitSend: aSend
   | aBlock |
   aSend message = #ifTrue:
      ifTrue: [ ↑ self compileSendIfTrue: aSend ].
   aSend message = #ifFalse:
      ifTrue: [ ↑ self compileSendIfFalse: aSend ].
   aSend message = #ifTrue:ifFalse:
      ifTrue: [ ↑ self compileSendIfTrueIfFalse: aSend ].
   aSend message = #ifFalse:ifTrue:
      ifTrue: [ ↑ self compileSendIfFalseIfTrue: aSend ].
   aSend message = #to:do:
      ifTrue: [ ↑ self compileSendToDo: aSend ].
   aSend message = #to:by:do:
      ifTrue: [ ↑ self compileSendToByDo: aSend ].
   self compileSend: aSend.
```

By using some simple tests certain special message sends are rewritten. In this case we have a look at the specialized `ifTrue:` message.

```
compileSendIfTrue: aSend
   | aBlock |
   aBlock := aSend at: 1.
   aBlock isScoped ifTrue: [ ↑ self compileSend: aSend ].
   aSend receiver accept: self.
   self append: #'send_ifTrue_' with: aSend.
   self compileBlock: aBlock.
   self code addLast: aBlock
```

First we check if the block passed as an argument to the `ifTrue:` message is really a block and if it is scoped. If it is not a block the standard protocol is followed and a full `ifTrue:` message is sent to the receiver. If the block has captured variables we cannot directly invoke its code either. Since a block context needs to be allocated we follow again the standard protocol. It is in these two non-optimal cases where a normal send object is used to follow the standard protocol by sending a message. The fallback mode is important to make the full Smalltalk MOP work properly. For instance in Pharo the special bytecodes handling the boolean messages are not aware of other types than booleans and fail on customized booleans. This constraint on the evaluated code works in most cases but defeats the purpose of a dynamic object-oriented system. Objects with compatible behavior should be interchangeable but with booleans in Pharo this is not the case. Hence the opcodes used in Pinocchio make sure the full protocol is supported. The following example shows that we first check for the known cases `true` – evaluate the block; `false` – do nothing; and the fallback where the `ifTrue:` message is sent to whatever type the receiver is.

```
1  OPCODE(send_ifTrue_)
2     Optr bool = PEEK_EXP(0);
      if (bool == true) {
4        Block block = (Block)get_code(pc + 2);
         pc += 3;
6        POKE_EXP(0, current_env());
         push_code(block->threaded);
8     } else if (bool == false) {
         POKE_EXP(0, nil);
10       pc += 3;
      } else {
12       Send send = (Send)get_code(pc + 1);
         push_closure(pc + 2);
14       Class_normal_dispatch(bool, send, 1);
      }
16 END_OPCODE
```

In case of the `send_ifTrue_` opcode almost no performance is lost by supporting the full MOP. But even if the additional fallback would impose an overhead it has to be supported nevertheless. All optimizations done by the underlying compiler have to be fully transparent to the language run on top even though the compiler is by nature closely connected to the VM and the language run on top. Nevertheless all optimizations in a dynamic language have to be transparent. If general optimizations start to constrain the language principles a language will evolve only into the direction of the highest performance gain, which is contradictory to the principles for instance of a Smalltalk with all its dynamic message sends and typeless code. The only way to be in harmony with the goals of a Smalltalk in particular, or a dynamic high-level language in general is to introduce optimizations in a non-intrusive way. Only this way the semantic assumptions of the language are kept throughout execution and optimization.

Next to special optimized opcodes for looping and branching there are versions for pushing the most common constants. Namely the constants `nil`, `0`, `1`, `2`, `true` and `false` are handled by special opcodes which directly push the elements on the stack:

```
1  #define PUSH(name, value) OPCODE(push_##name) \
2     PUSH_EXP(value);\
      pc = pc + 1;\
4  END_OPCODE

6  PUSH(nil, nil)
   PUSH(0, smallInt_0)
```

```
8 PUSH(1, smallInt_1)
  PUSH(2, smallInt_2)
10 PUSH(true, true)
  PUSH(false, false)
```

The choice for these hard-coded push opcodes was guided by the existing choice of Smalltalk and observing the integer usage pattern. Although there could be a potential bigger choice for constant opcodes – taking the whole integer range for example – they greatly affect the size of the binary. Thus the speed could suffer from degenerated memory access and CPU caching in the resulting binary. Furthermore the speedup achieved by this optimization is minimal as only few indirections are avoided.

4.4.1 Evaluation

As a first evaluation we have a look at the speed impact of the specialized send opcodes. The results are shown in Table 4.5 and the Figure 4.16. The impact is again varying from benchmark to benchmark. Nevertheless both the parser and the dictionary benchmarks show around the same slow-down. The Fibonacci benchmark is around four times slower than with the optimization which can be traced back to the simple structure of the benchmark amongst other heavily relying on boolean branching. In general we can say that the overall speedup introduced by the set of special send opcodes is around the factor two – quite an impressive result for the simplicity of the implementation.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	6.6073(16)s	2.9184(15)s	2.264
Dictionary	0.3901(43)s	0.1707(17)s	2.285
Fib 31	3.3671(11)s	0.82383(79)s	4.087

Table 4.5: Benchmark evaluation without any special opcodes

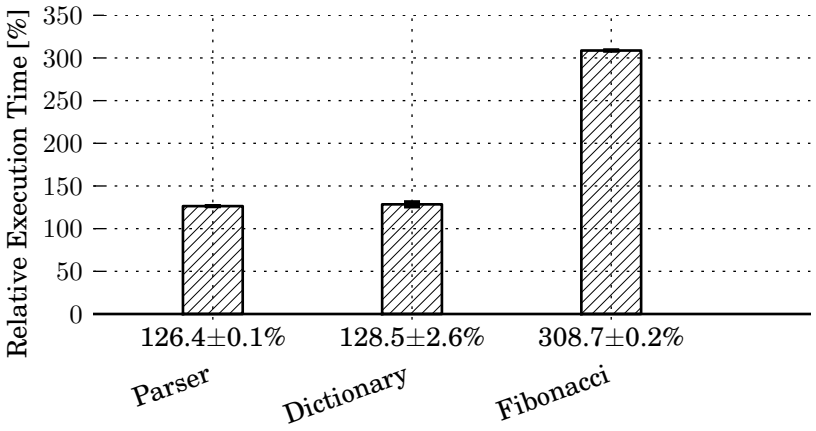

Figure 4.16: Performance impact with special opcodes disabled

As a next step we have a look at the impact of the specialized boolean sends in detail. The results are presented in the Table 4.6 and the relative impact in Figure 4.17. What has been previously assumed about the Fibonacci benchmark is clearly visible here – it heavily profits from the optimized branching in opcodes. The overall impact on the two other benchmarks is less significant. The dictionary benchmarks show almost no

use of these boolean branch sends and thus show an overall slowdown of only around 10%. The rather algorithmic code of the parser benchmark is with 50% more affected.

Benchmark	Time without Optimization	Time with Optimization	Rato
Parser	4.3000(20)s	2.9184(15)s	1.473
Dictionary	0.1843(18)s	0.1707(17)s	1.080
Fib 31	3.3943(14)s	0.82383(79)s	4.120

Table 4.6: Benchmark evaluation without special boolean opcodes

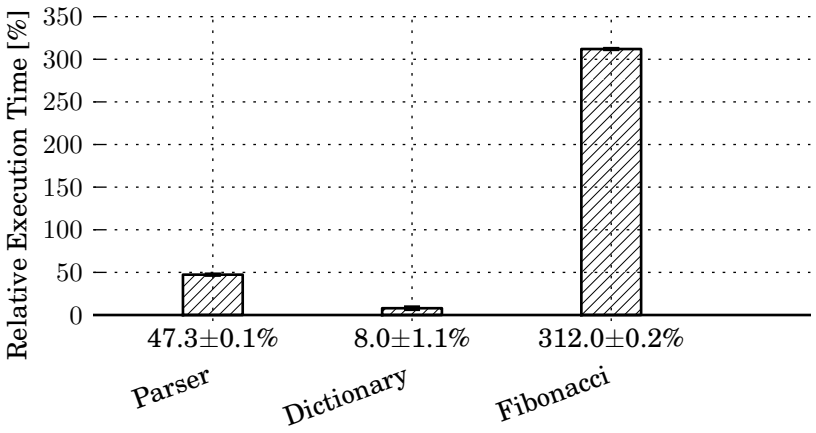

Figure 4.17: Performance impact with special boolean opcodes disabled

To justify the results of the first evaluation having all special send opcodes disabled we have to run the two loop optimizations separately. First we removed the simple `to:do:` optimizations resulting the in values in Table 4.7 and the Figure 4.18. Of course there is no impact on the Fibonacci benchmark since we have shown that most of the speedup there results from the boolean optimizations. The parser is again affected since it uses simple loops over collections. Dictionaries are less affected by this optimization since there the most of the looping is done in buckets but only in steps of two.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	4.3335(13)	2.9184(15)s	1.485
Dictionary	0.1807(19)	0.1707(17)s	1.058
Fib 31	0.8252(15)	0.82383(79)s	1.002

Table 4.7: Benchmark evaluation without special `to:do:` opcode

The second send optimization for iterations in steps other than one, done by the `to:by:do:` message on integers mostly affects the dictionary code. To find an element identified by a particular key all the existing keys in the bucket have to be checked. Since the values in the bucket are stored pair-wise – first the key then the value – only every second item in the bucket is a key. Thus iterating over the values happens in steps of two and not just one. Hence the different results whether the `to:do:` or the `to:by:do` is deactivated.

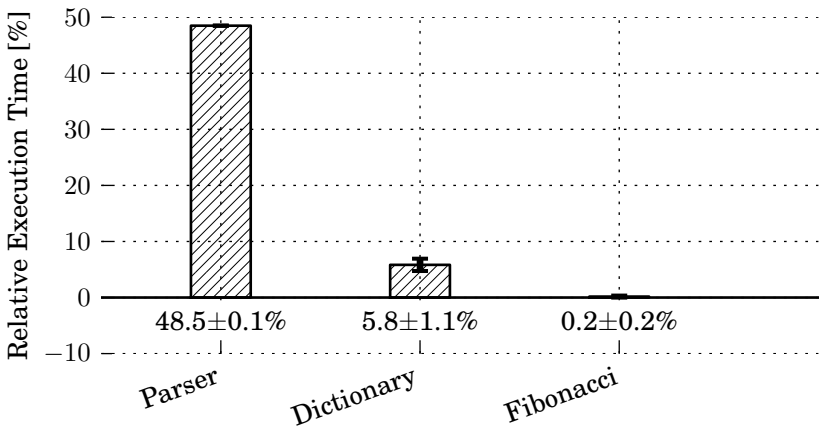

Figure 4.18: Performance impact with special `to:do:` opcode disabled

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	2.8863(16)	2.9184(15)s	0.989
Dictionary	0.2642(19)	0.1707(17)s	1.548
Fib 31	0.8768(10)	0.82383(79)s	1.064

Table 4.8: Performance impact with special `to:by:do:` opcode disabled

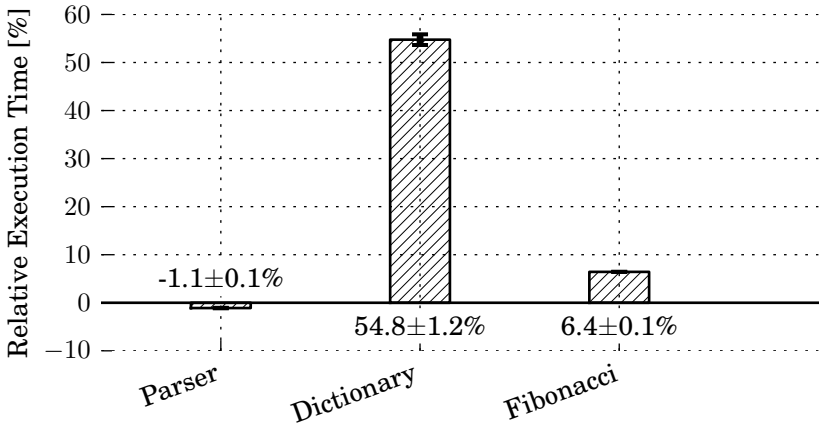

Figure 4.19: Benchmark evaluation without special `to:by:do:` opcode

The results about the opcode optimizations discussed so far only cover the optimization on the existing opcode infrastructure. The impact of the opcode evaluation scheme over the old AST version has not been evaluated yet. During the start of the optimization period we decided to track the progress on the minimal Fibonacci benchmark. Hence we setup a speedcenter which regularly runs the benchmark and uploads to the result to a website. The project was originally created for the PyPy project[1]. Figure 4.20 shows only a small number of data points. It seems that Pharo has not been designed to run in headless mode, since merge conflicts raised window pop-ups which had to be resolved manually. Nevertheless the graph shows a raw overview of the speed

[1] `http://speed.pypy.org`

of Pinocchio.

The initial speed in the early days of the graph is still achieved by a slightly optimized version of the AST visitor. The following data points residing at almost the same value of 2.4 seconds were several attempts to optimize the AST evaluator further. Since all the micro optimizations were not very successful a different path had to be taken. The only way to achieve further speedup was to change the evaluation scheme. Hence we implemented an opcode based evaluator. Thus the following steep improvement was the result of this paradigm change. Roughly spoken the opcode evaluation improved the speed by a factor of eight.

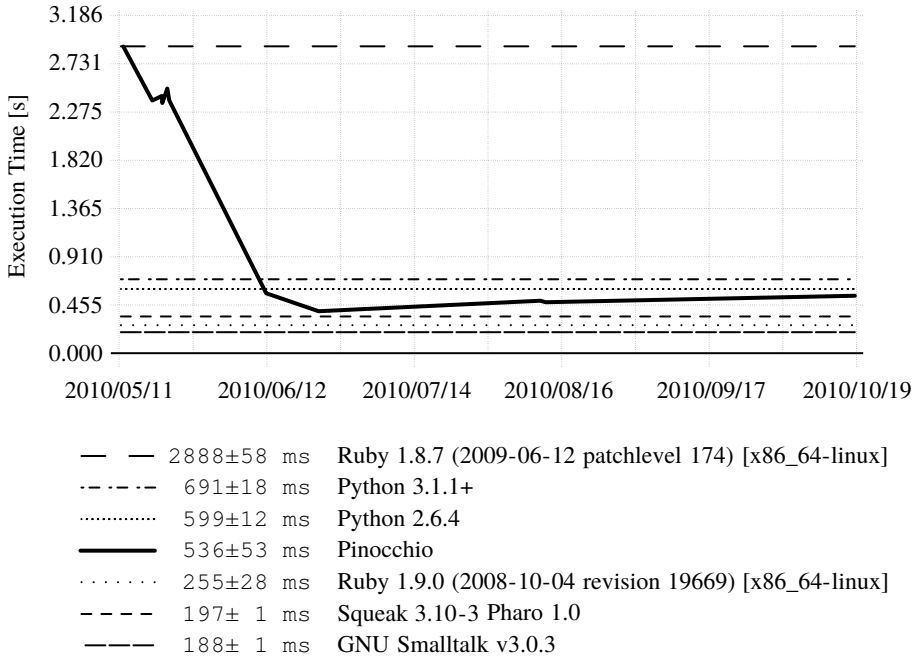

Figure 4.20: The Pinocchio speedcenter. Evolution of the Fibonacci benchmark over time

4.4.2 Issues

The overall speedup introduced using opcode evaluation is significant – even with a very simple compiler. With the shift from the pure AST evaluation to the opcode based version we could observe an increase in the size of the resulting binary since we still keep the AST code for inspection and possible refactorings. The resulting binary of the core Pinocchio is around 2.5MB which is still very little on a modern system. A pure opcode system would reduce the size of the binary but also drop support for custom interpreters as they rely on AST nodes for evaluation. There is basically only one valid argument against opcodes and in favor of bytecodes. Using bytecodes makes a language more system independent. Since the opcodes point directly to the address where the function is stored there is no way to make one single version available on different platforms. To get system independent versions you could consider to use an intermediate bytecode-based format which gets dynamically translated to a threaded code. This way you could profit from the platform independence of bytecodes and the speed advantages of threaded code.

4.5 Caching Integers and Characters

In Pinocchio and in Smalltalk, integers are seen as normal objects from the user point of view. Thus performing numeric calculations comes with a significant overhead. For each low-level integer we have to create a full object instance. However there are ways to reduce the memory footprint of full objects by using tagged integers. One bit of each pointer is reserved to indicate whether the object is an integer or not. Thus for a normal object the pointer points to the location of a valid memory location. However for integers the pointer value itself is used as the final integer value. And since integers are immutable there is no need to have a corresponding memory location for storing instance variables. A single integer instance known to the VM suffices. Another way to reduce the cost of creating integers is to cache the most commonly used integers. Hereby a certain range of integers is precalculated. The following code excerpt shows the added indirection for integer creation.

```
1 SmallInt new_SmallInt(long value)
2 {
     if (CACHE_LOWER <= value && value < CACHE_UPPER) {
4       return SmallInt_cache[value];
     }
6    return raw_SmallInt(value);
  }
```

Instead of directly returning a new instance, precalculated integers are returned whenever the value lies within a given range. The current caching range is from -1 to 1024 which is proven sufficient in the following evaluation section. The same principle was applied to characters where we cache the ascii range.

4.5.1 Evaluation

The graphs on Figure 4.21 and Table 4.9 show the impact of disabling the integer caching. Obviously the Fibonacci benchmark is heavily affected by the speed of integer creation. The speedup achieved for this particular benchmark with integer caching enabled is around the factor 3 whereas the other two benchmarks are less affected. Still both the dictionary and the parser benchmark can benefit with around 30% speedup.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	4.0327(18)s	2.9184(15)s	1.382
Dictionary	0.2158(19)s	0.1707(17)s	1.264
Fib 31	2.84929(75)s	0.82383(79)s	3.459

Table 4.9: Benchmark evaluation without a SmallInt cache

The results presented here are in comparison to the default integer caching for a range from -1 to 1024. The boundaries were initially determined experimentally. To fully assure that this range makes sense we tracked the integer usage over the three benchmarks. The graph on Figure 4.22 shows that there is an almost perfect match for the Parser benchmark. The parser mostly uses the integers in the range of -1 and $2^{11} = 2048$. Generally there is a certain percentage of usage in the very high ranges resulting from using the C-level pointer values directly as hash values.

The results from analyzing the impact of the character cache on the three benchmarks is presented in Table 4.10 and Figure 4.23. There is no measurable impact on the dictionary and the Fibonacci benchmarks since there are no characters involved during the whole calculation. But for the parser benchmark we see a 6% performance loss. There would be even a bigger performance gap if the parser would not have

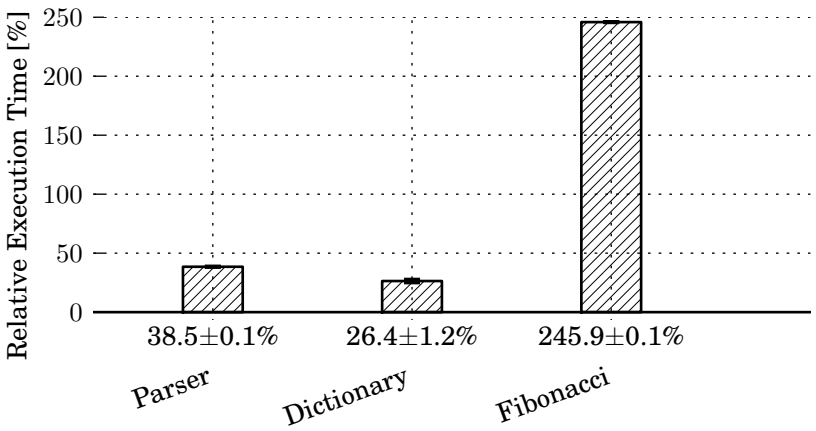

Figure 4.21: Performance impact with disabled SmallInt cache

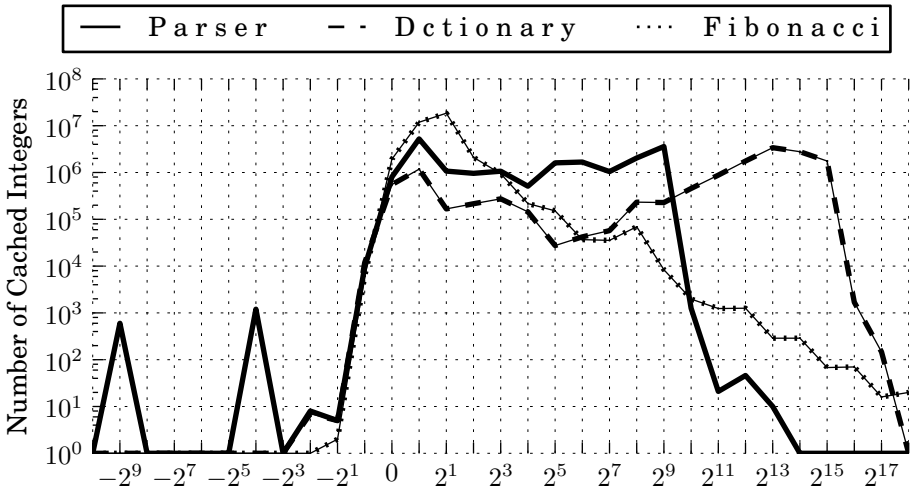

Figure 4.22: Histogram of cached Integers for different Benchmarks

been refactored to almost never use characters. To compare the speed of the Pinocchio parser with the one from Pharo several optimization decisions were taken to increase performance on Pharo. One of them was to use integer instead of character comparison as it is significantly faster under Pharo. Since strings internally consist of an array of C-characters there are almost no characters needed and the integer value for a character can be directly extracted with a native.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	`3.1038(15)s`	`2.9184(15)s`	`1.064`
Dictionary	`0.1700(17)s`	`0.1707(17)s`	`0.996`
Fib 31	`0.81594(72)s`	`0.82383(79)s`	`0.990`

Table 4.10: Benchmark evaluation without a character cache

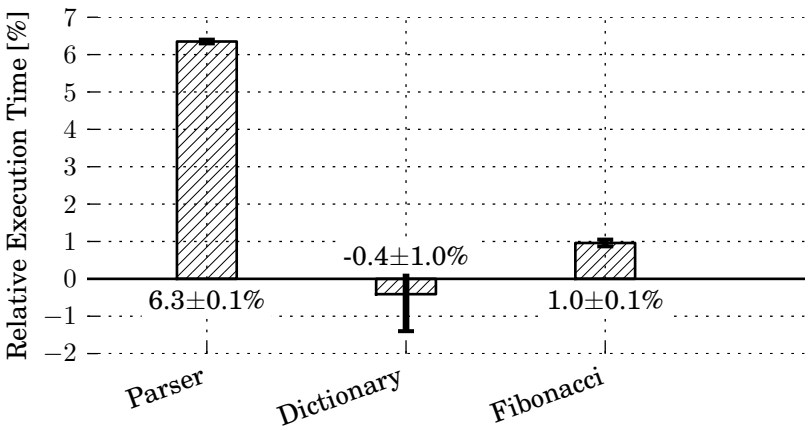

Figure 4.23: Performance impact with disabled character cache

4.5.2 Issues

The evaluation of the SmallInt cache has shown a significant speedup. On the other hand the addition of a character cache did not significantly improve performance which is also influenced by the choice of benchmarks. They almost never use characters. Using caches is a simple way to increase performance but on the downside might pose a significant memory overhead. In the case of the SmallInt cache roughly $1024 \cdot 8Bytes \cdot 3 = 24$KB is used to cover the most common cases on a 64Bit machine – nowadays a neglectable amount. In every case it is important to estimate a proper range of values. For instance it would make no sense to precalculate the full range of the SmallInts as it would exhaust the addressable memory on every system. Furthermore it is important that the cache can be accessed with a minimal overhead. In the case of such simple objects as SmallInts it might even pay off to generate the values on the fly rather than precalculate them. The additional check if an element has already been stored would have a negative impact – especially when more complex operations than a simple `NULL`-check is involved. But the cache access time needs to be kept in relation to the object creation size and yield a positive result to eventually provide a speedup.

4.6 Inline Caches

One reason why dynamically typed languages run slower than statically typed ones is the high number of dynamic message sends. . One way to improve performance for send sites is to use inline caches (IC)[2]. Instead of dynamically looking up a method a class method pair is stored in the IC and used for directly dispatching. Inline caches work well when there are no changing types for the receiver. In case the number of different types is limited, polymorphic inline caches (PIC)[6] can help. In contrast to single lookup caches PICs can store the recent method lookups for more than just a single type.

Supporting inline caches in Pinocchio is rather simple as mainly the dispatching code is affected. Considering the original dispatch code in the following example:

```
1 void Class_normal_dispatch(Optr self, Send send,
                      uns_int argc)
3 {
    Class class = HEADER(self);
5   assert_class((Optr)class);
```

```
    Symbol msg = (Optr)send->message;
    assert0(msg != nil);

    PUSH_EXP(send);
    Class_do_dispatch(self, class, msg, argc,
                  T_Class_dispatch);
}
```

First we extract the class from the current `self`, then the message selector. Finally the send object is pushed onto the stack and a class dispatch is performed. The `Class_do_dispatch` performs the selector lookup through the class hierarchy and might perform the `doesNotUnterstand` send if the selector is not found. To support inline caches in this code we shortcut the `Class_do_dispatch` by first checking if we already dispatched on the specified class and selector pair. The Following code shows the current version of the dispatch function including support for inline caches.

```
void Class_normal_dispatch(Optr self, Send send,
                     uns_int argc)
{
    Class class = HEADER(self);
    Array cache = send->cache;

    if ((Optr)cache != nil) {
        Optr method = InlineCache_lookup(cache, class);
        if (method) {
            return invoke(method, self, argc);
        }
    } else {
        send->cache = new_InlineCache();
    }
    assert_class((Optr)class);

    Optr msg = (Optr)send->message;
    assert0(msg != nil);

    PUSH_EXP(send);
    Class_do_dispatch(self, class, msg, argc,
                  T_Class_dispatch);
}
```

Now instead of directly continuing with the selector-lookup we lazily use the inline cache. Caches are currently created lazily to increase the startup speed of the system as there are several thousand send sites to be initialized. If a send with a inline cache is found we perform a lookup directly in the inline cache rather on the class hierarchy by calling the `InlineCache_lookup` function in C. This function checks if the inline cache as already encountered the incoming class as a message receiver. For single inline caches the lookup is a single pointer equality check. For polymorphic inline caches a linear search is performed – as demonstrated in the following code example:

```
Optr InlineCache_lookup(Array cache, Optr class)
{
    int i;
    for (i = 0; i < GET_SIZE(cache); i += 2) {
        if (cache->values[i] == class) {
            return cache->values[i+1];
        }
```

```
    }
9   return NULL;
}
```

Once the class is found in the cache the corresponding method is returned and can be invoked directly. In order to fill the inline caches properly a second change is necessary in the lookup code of Pinocchio. Each time a lookup is performed the resulting method has to be stored in the inline cache.

4.6.1 Evaluation

Next to running the three default benchmarks we additionally track the usage of inline caches to validate the performance impact of polymorphic inline caches. By using a small DTrace script we can precisely see how many cache hits and misses occur during a benchmark. Furthermore we can even accurately count the cache misses per selector and class. Enabling DTrace probes of course has a slight performance impact as a small helper code needs to be activated on each cache access and message send. Hence we use a different version for the tracing than for measuring the time spent for each benchmark.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	3.0796(12)s	2.9184(15)s	1.055
Dictionary	0.1683(18)s	0.1707(17)s	0.986
Fib 31	0.9421(16)s	0.82383(79)s	1.144

Table 4.11: Benchmark evaluation without a polymorphic inline caches but simple inline caches

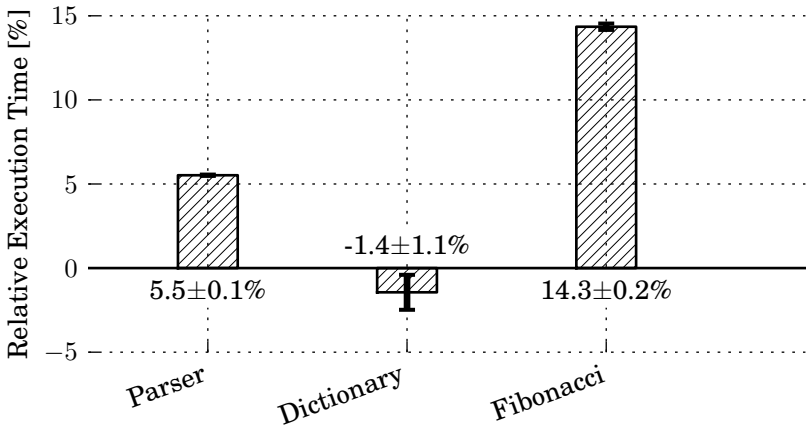

Figure 4.24: Performance impact with disabled polymorphic inline caches

Table 4.11 and Figure 4.24 show the outcome of the three default benchmarks when polymorphic inline caches are disabled but single inline caches are still in use. Table 4.13 shows the number of cache misses and cache hits. Table 4.12 shows the same values with polymorphic inline caches showing a hit rate of very close to 1 for all three benchmarks. Even though only single inline caches are used the hit rate is amazingly high except for the dictionary benchmark. Thus the results can only be affected in the percent range. The code used in the benchmarks does not have deep class hierarchies and thus the overhead of a single lookup is not high. The dictionary benchmark has

Benchmark	Cache Misses	Cache Hits	Hit-Rate
Parser	6.051E+04	4.985E+07	0.99879
Dictionary	1.877E+03	6.199E+07	0.99997
Fibonacci	7.500E+02	3.191E+07	0.99998

Table 4.12: Cache misses with polymorphic inline caches for the three default benchmarks

Benchmark	Cache Misses	Cache Hits	Hit-Rate
Parser	2.808E+06	4.710E+07	0.94373
Dictionary	3.414E+07	2.786E+07	0.44935
Fibonacci	1.267E+05	3.178E+07	0.99603

Table 4.13: Cache misses with normal inline caches for the three default benchmarks

only such a high miss rate for single inline caches as the benchmarks consists of two different parts. First the benchmark is run with normal `PDictionary`, then the base benchmark (see Section 3.2) is run over the same code base. But this time a dummy dictionary implementation is used to avoid any overhead introduced by the data structure itself. Of course with single inline caches the second run of the benchmark will only hit already filled-in caches from the first run. Notably the caches are filled with the `PDictionary` classes and not the dummy dictionary implementation leading to the high number of cache misses without any significant effect on the final results.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	3.0703(12)s	2.9184(15)s	1.052
Dictionary	0.1589(15)s	0.1707(17)s	0.931
Fib 31	0.82532(62)s	0.82383(79)s	1.002

Table 4.14: Benchmark evaluation without any inline caches

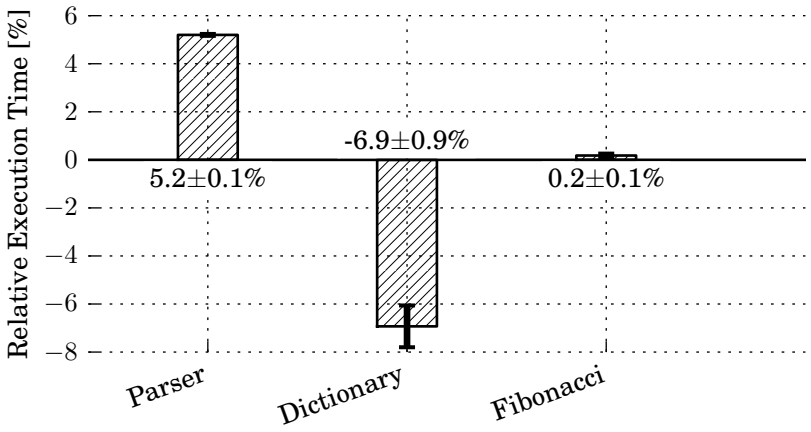

Figure 4.25: Performance impact with disabled inline caches

The following results in Table 4.14 and Figure 4.25 are created by disabling all types of inline caches. Especially in the parser benchmark which should be mainly

affected by the presence of inline caches there is no significant slowdown visible. This leads to the assumption that inline caches – polymorphic or not – generally have only little impact on the performance. From an optimization point of view this is not a favorable property. But since inline caches can be further used for JIT compilation the non-negative performance impact again is an advantage. Counting polymorphic inline caches can be used to track the types of each send site. Next to counting the occurrence it pays off to normalize the obtained type histogram and drop the types with low probabilities. This way the JIT can react to changes in the code with recompiling parts where new types occur.

Even though both types of benchmarks with simple inline caches in Table 4.11 and without any inline caches in Table 4.14 did not present significant changes, we were still interested in which parts of the system fail on polymorphic inline caches. Table 4.15 shows the selectors for the 10 topmost cache misses. The `initialize` cache misses has its seeds in a megamorphic send site namely the `new` method. In most cases `new` is never overridden thus for most object creations the very same method is used. But in many cases the `initialize` method is overridden to perform custom actions on object creation. Thus the `initialize` send in the `new` method is not bound to a small set of specific types and thus can be considered as a megamorphic send site. In this case polymorphic inline caches do not provide any advantage but rather a small overhead. Instead of directly looking up a method dynamically the cached list of types is traversed first – in most cases unsuccessfully.

Message	Cache Misses
`initialize`	26664
`parseOn:`	25575
`asParser`	1602
`=`	896
`class`	892
`asChildParser`	761
`initialize:`	661
`expression`	141
`identifier`	140
`primary`	120

Table 4.15: Message names and the number of the 10 topmost cache misses for the parser benchmark using polymorphic inline caches.

4.6.2 Issues

We have shown that single and polymorphic inline caches do not provide a significant speed improvement. This low impact might be cause by the flawed nature of the benchmarks used. Only the parser benchmark provides a setup with several classes which inherit from each other. But even then the class hierarchy is comparably flat. The maximum depth is 4, achieved by the AST nodes. Nevertheless, even with the limited applicability of these benchmarks several issues can be observed with the different types of inline caches. Polymorphic inline caches work very well on the code presented, resulting in less than 1% cache misses as shown in Table 4.12. Combined with Table 4.15 we can deduce that only very few send sites are megamorphic. However for megamorphic send sites the inline caches pose some overhead as most type lookups will fail. For the simple implementations used in the Pinocchio the overhead is negligible. However the overhead grows with the complexity of the inline caches.

For instance to render an inline cache fully usable for JITs it is required to count the frequency of each encountered type. Furthermore the caches need to be cleared regularly to allow for changes. In these cases it might be of advantage to define a threshold of type misses and cache misses upon which the caching would be disabled in order to reduce the overhead. Another issue arising from inline caches is their timeliness. In a dynamic system such as Smalltalk methods can be removed, added or changed at runtime requiring inline caches to be flushed or to be updated. At the moment Pinocchio does not have such a feature. Once filled, inline caches keep their information forever. In the Pharo VM this problem is solved by having only one global cache table storing the class, selector and send site . Hence updating the cache is simple as there is only one centralized storage place. For Pinocchio two procedures could be considered. The first solution requires a full traversal of the AST nodes to find all affected caches on method modification. Another possibility would be to store a reverse link on each Class to the send sites. This way the affected send sites can be easily deduced from the class to which modified method belongs to. The first solution will be rather slow due to a potentially high number of AST nodes. The latter solution shifts the load to the memory side and requires more changes to the sources. The important question is how frequent methods are changed, added or deleted in such a system. If there is only a low probability for method changes the overhead can be of course considerably high and thus in favor of the first solution. In general inline caches only make sense in the anticipation of a JIT where type information is required to create fast specialized versions of methods or traces. If this is not the goal the inline caches mostly are an unnecessary engineering task. In consideration of a dynamic system with changing classes and methods it might make sense to even drop support for inline caches fully.

4.7 Unwrapping Values

Smalltalk features different special layout types to increase performance. One commonly used layout is the one for array objects. Arrayed objects have a reserved size slot and a fixed number of elements stored. This can be of advantage in several ways. First of all there is no need to add an instance variable to hold the elements and second there is no need to create a size instance variable. By creating arrayed objects the elements can be directly accessed using the `at:` and `at:put:` selectors. In the case of Pharo these two selectors are implemented as natives which directly check the boundaries of the passed-in index. Hence the VM is fully aware of the data structure and the access can be optimized. Next to just implementing the accessor methods directly in C, the structure itself can be adapted to better serve the low-level code. One possibility is to store the size value directly as a C-level integer instead of a high-level SmallInt. Thus there are no checks or conversions needed on C-level when accessing or checking the boundaries. However accessing the size from within application-level code requires a conversion back to the high-level integer representation. Whether it is an advantage to store the unwrapped value in the data structure is the subject of mainly two factors. Since the size of an array does not change at runtime the overall performance gain depends on the ratio of size accessing and data accesses. If the code requires to know the size of an array at many places the wrapping of the low-level size value is an overhead. On the other hand if the number of accessor message sends outnumbers the size accesses storing the low-level size value is an advantage.

As an example of an arrayed object the `PBlock` AST node of Pinocchio is presented in the following code excerpt.

```
PNode variableSubclass: #PBlock
   instanceVariableNames: 'params locals threaded'
   classVariableNames: ''
   poolDictionaries: ''
```

```
    category: 'Pinocchio-Kernel-AST'
```

In Pharo the special layout is denoted by using `variableSubclass:` instead of `subclass:` as the selector to create a new subclass. `PBlock` has three instance variables – `params`, `locals` and `threaded` – but no explicitly mentioned variable for storing the AST nodes of the body. Since `PBlock` is declared to be arrayed, values can be directly stored on the object using the native accessors. The following example illustrates this behavior by creating an `PBlock` AST node of size 1 and storing a constant object in the body:

```
block := PBlock new: 1.
block at: 1 put: false_constant.
block size should be = 1.
```

By using an arrayed object we can simplify the code in Smalltalk. The basic behavior for `at:`, `at:put:` and `size` work out of the box. Furthermore we can get a small speedup as the VM is aware of the basic data structure. Let us examine the most basic arrayed object the Array itself. Its C-level data structure is the following:

```
struct Array_t {
    int size;
    Optr values[];
};
```

The size itself is stored as a raw integer which makes writing natives slightly easier and a bit faster. To validate the performance increase we have to be able to easily interchange the two different size formats in the C sources. To do so we introduced three different macros in Pinocchio: `GET_SIZE` and `SET_SIZE` for accessing the size of an arrayed object and `ARRAY_SIZE_TYPE` for the type of the size property. Using a single switch `ARRAY_WRAPPED` we can easily change the type at compile time. The following code shows the different macro implementations:

```
#ifdef ARRAY_WRAPPED
    #define GET_SIZE(array) \
          ((Array)(array))->size->value
    #define SET_SIZE(array, value) \
          ((Array)array)->size = new_SmallInt(value);
    #define ARRAY_SIZE_TYPE SmallInt size
#else //ARRAY_WRAPPED
    #define GET_SIZE(array) \
          ((Array)(array))->size
    #define SET_SIZE(array, value) \
          ((Array)array)->size = (value);
    #define ARRAY_SIZE_TYPE uns_int size
#endif //ARRAY_WRAPPED
```

Both for `GET_SIZE` and `SET_SIZE` one pointer indirection more is necessary to access the raw integer from C-level. By using tagged integers instead of real objects this could be replaced with a bit mask operation checking if the lowest bit is set and thus denotes an integer. The previously presented C-level structure for arrays can be replaced by the following adapted version:

```
struct Array_t {
    ARRAY_SIZE_TYPE;
    Optr values[];
};
```

The `ARRAY_SIZE_TYPE` macro will expand to the appropriate type – either a raw integer or a high-level SmallInt. The following subsection will cover the performance difference by using the two different types for size on arrayed objects.

4.7.1 Evaluation

Table 4.16 and Figure 4.26 show the impact of using wrapped integers for the size value of arrayed objects. The results show no significant speed difference which is not what was expected. Although the parser benchmark shows some small slowdown it is still very little compared to the variation of the results. It seems that using SmallInts does not pose a significant overhead. We expected that the double dereferencing in the C-level for accessing the native value of a SmallInt would introduce more overhead in total.

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	2.9702(16)s	2.9184(15)s	1.018
Dictionary	0.1731(18)s	0.1707(17)s	1.014
Fib 31	0.81136(72)s	0.82383(79)s	0.985

Table 4.16: Benchmark evaluation with wrapped SmallInt for arrayed objects

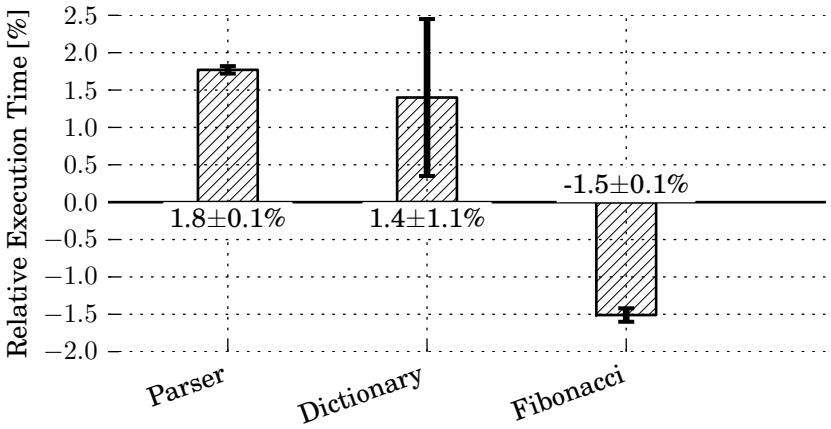

Figure 4.26: Performance impact with wrapped SmallInt for arrayed objects

The only reason for the positive results must lie in the creation of SmallInts. In a system like Pharo using tagged pointers to mark integers it is comparably cheap to create a bulk of new integers. But in Pinocchio full objects need to be created which should be fairly expensive if used in frequent places like arrayed-objects. Thus the caching of SmallInts discussed in Section 4.5 must be the main source for these unexpected results. To prove our hypothesis we run the same benchmarks with wrapped values for the size of arrayed objects but this time without a SmallInt cache. Finally the results in Table 4.17 and in Figure 4.27 show the expected behavior. Comparing these results with the ones where only the SmallInt caches are disabled in Table 4.9 and Figure 4.21 a significant difference can be observed. When additionally using wrapped integers for arrayed objects the parser benchmark is another 60% slower, the dictionary benchmark around 50% and the Fibonacci benchmark an additional 140%.

4.7.2 Issues

Using special layout objects to improve speed seems to be a reasonable choice in a dynamic language VM. Doing so can improve the speed of the natives in a system as less unwrapping and wrapping of values is needed. The choice for the special layouts

Benchmark	Time without Optimization	Time with Optimization	Ratio
Parser	5.7054(22)s	2.9184(15)s	1.955
Dictionary	0.2921(28)s	0.1707(17)s	1.711
Fib 31	0.81826(30)s	0.82383(79)s	4.635

Table 4.17: Benchmark evaluation with wrapped SmallInts for arrayed objects and no SmallInt cache

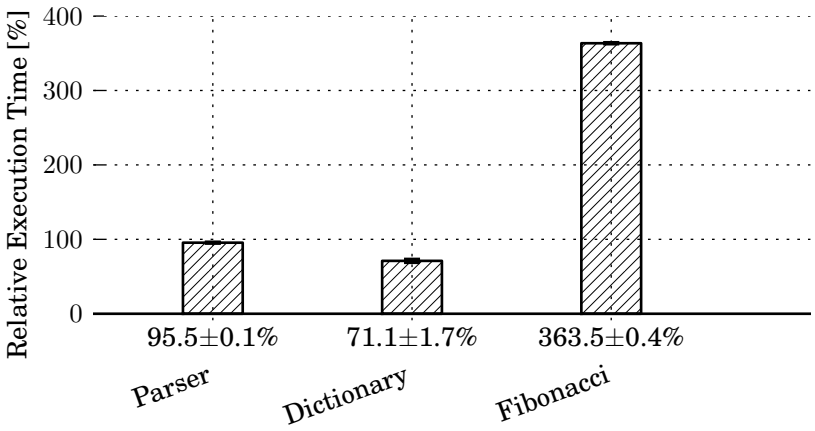

Figure 4.27: Performance impact with wrapped SmallInts for arrayed objects and disabled SmallInt cache

depends on the usage of natives. If there is only a limited number of natives in use it makes no sense to clutter the VM code with hard-coded checks for certain layouts. But if the layout facilitates common operations like the `at:` and `at:put:` in the arrayed layout there is room for a performance increase. In the case of arrayed objects we could show that in combination with a sufficient SmallInt cache there is no need for this type of layout. In combination with the cache, objects can be used with the default layout and do not suffer from performance loss in Pinocchio. Hence there is no need to further keep the arrayed layout in Pinocchio. Although you can argue in favor of the special layout type if there are many natives depending on it. But for the arrayed-objects there is only a limited set of natives which are faster than normal Smalltalk code. Since the type of the elements stored in the arrayed object are not known upfront most operations besides accessors require sending messages back to each element. This renders most natives useless and mainly increases the C-level codebase which is generally harder to maintain. In short, unwrapped values only pay off if there is a sufficient number of natives which can directly use these values.

4.8 Future Work

The current version of Pinocchio still suffers from a full base system. Although we can easily export parts of the existing Pharo images it involves several cleanup steps to assure the usage of the proper protocols. So far we paid attention to cleanup the exported code whenever possible. This is done by only exporting a minimal set of methods necessary to make the code run. Refactoring the code and introducing new protocols is somewhat cumbersome as it makes it harder to run the same code on top of Pharo and Pinocchio. However this is still required for as long as the base system is not mature enough to support decent file based sources or creating a new image. An

important step would be to define a format to write Pinocchio in files. So far there is only a generic syntax for creating new classes and installing methods. A compatible solution would be to follow the file format of the non image based GNU Smalltalk[2]. The following step to complete the base system would be to implement bindings for a GUI library such as GTK or Qt and start building the tools necessary for development.

The current version of Pinocchio can be seen as partially image-based since the bootstrapping is hard-coded in the image. However it would be nice to have support for images to improve startup time. For instance the core classes could be stored in the bootstrapped state in an image avoiding the necessity of creating them over and over again on each Pinocchio launch. The same can be said for library code, instead of compiling the code on each load it could be stored after the first compilation in an image. In combination of the opcode execution scheme the question arises whether or not the image format should be platform independent.

4.8.1 Compiler

One way to increase performance of Pinocchio is to extend the compiler and to make it more intelligent. The current compiler could be extended with Static Single Assignment (SSA)[1] and Three Address Code (TAC) transformations to support further optimizations. Using SSA would allow the compiler to perform constant folding and constant propagation. Although this might be applicable only in rare cases with the current programming style in Smalltalk. Most methods have only very small bodies and rather send various messages than directly having the code inline. In combination with automatic inlining and thus growing body sizes optimizations can be of more help.

To support such intermediate formats the current compiler needs to be rewritten completely. At the moment the compiler consists of three phases. First a Smalltalk parser creates Smalltalk ASTs which then are converted in a second step to Pinocchio AST nodes by a simple visitor. In the third and last step the Pinocchio AST is flattened out to opcodes – again by a visitor-based implementation. Conceptually the main difference between the Smalltalk and the Pinocchio code are the direct support for message cascades in Smalltalk, which are converted to passing the receiver of a cascade to a block in the current version of Pinocchio. For instance the basic following basic example,

```
result := (Array new: 2) at: 1 put 2; yourself
```

is translated to code without cascades using an intermediate block:

```
result := [:value|
          value at: 1 put 2.
          value yourself] value: (Array new: 2)
```

This way of handling cascades is easy and straight-forward to implement but not very efficient. It requires allocating an additional block, a block activation with an additional message send. A more optimal form should use a temporary variable to avoid the block at all:

```
tmp := (Array new: 2)
tmp at: 1 put: 2.
result := tmp yourself
```

Of course this is only a very simple example of how to improve our compiler and with respect to the frequency of cascades with very limited performance increase. More interesting would be to support inlining and then optimize the resulting code. However inlining only makes sense if there is type information available. One way would be to use runtime information and track the types on each send site. Another approach is to

[2] http://smalltalk.gnu.org/

have type-aware AST nodes which could handle enforced types. In either case there is additional compiler support needed with a pluggable architecture.

4.8.2 Runtime Optimizations

Depending on new intermediate formats like SSA and TAC in the compiler it would be possible to drop most of the current VM's stack usage – currently a significant overhead. Pushing all calculated values first onto the stack and then later on removing them again involves unnecessary decrementing and incrementing of the stack pointers. Although this is a fairly cheap operation it can happen multiple times in a single AST node evaluation. So instead of pushing values temporarily on the stack, the stack frame can be used directly. Values for temporaries and instance variables can be stored directly in the proper location. To accomplish this change several changes to the evaluation strategy of Pinocchio are necessary. Furthermore the opcodes have to be changed in most cases to explicitly target the former AST visitor. Another possible optimization is to have a dedicated return register which prevents further stack operations.

As already mentioned in the previous Section 4.8.1 the use of runtime information can help to improve performance further. The main principle is to embed assumptions at runtime. To do so we need a way to track the type of message receivers. The use of inline caches – discussed in Section 4.6 – only helps to increase performance marginally but allows for easy tracking of the receiver types. At the moment we track the receivers in a very limited way and only fill in new types if there are still empty slots available. The proper behavior would be to count the occurrences of a type and then store the most often used ones. Next to counting there has to be a normalization over time to be able to adapt to new situations where other types are used. Once reliable type information is available we can start to recompile the methods. One approach is to start inlining other messages, the other would be to record an execution trace and compile that. Recording traces will require some changes to the VM if native support is required. Another way would be to create a tracing interpreter on top of Pinocchio. Although the stacked interpreter adds some overhead it would greatly simplify the complexity and improve maintainability as all the code could be written in Smalltalk. This tracing interpreter then needs to recompile traces to binary code when a certain threshold is hit to compensate for the additional evaluation overhead.

4.8.3 VM Optimizations

As mentioned in Section 4.4 direct-threading is faster than an indirect-threaded approach. At the moment we store directly a sequence of function pointers as the main evaluation data, compared to bytecodes or AST visitors a very fast and simple solution. But at the moment each opcode modifies a program counter and returns after finishing its calculations. By directly jumping to the following opcode address the overhead of calling a function and returning from it can be avoided. The ultimate step in this series of optimizations is to directly copy over the native machine code into each method body. This allows the branch prediction of the CPU to work more efficiently and avoids dispatch overhead of opcodes completely.

Furthermore several initial ideas need to be fully completed in future versions of Pinocchio. Stacked interpreters for instance are supported in a limited way but could be further analyzed and improved. Another issue is the usage of slots in the system. In the version discussed in this thesis slots are not yet supported. They are only used as a placeholder for the name of the instance variables and are used to determine the size of the object. However, the instance variable access is handled in the same way as in Pharo by directly indexing into the object's data. Future versions of Pinocchio should support sophisticated slots with their own state and which can modify the instance variable access. The ultimate goal is to have no overhead for default slots by compiling

away the intermediate slot objects and directly accessing the stored data. Of course this requires a much more elaborate compiler supporting basic dynamic changes to the AST at compile time.

5 Conclusion

This thesis sheds light on the different aspects of optimizations using the Pinocchio Virtual Machine. Chapter 3 starts by explaining the low-level details of benchmarking followed by presenting specific results in Chapter 4. The benchmark results in this thesis focus on optimizations useful in a high-level language VM. This is very different from low-level optimizations which would directly apply to C code. Although such optimizations are important on a local scale in the VM sources, they generally contribute only a small percentage to the final speed of a high-level language VM. The optimizations are dividable into two different abstract approaches. Non-transparent optimizations which require changes on the semantics of the interpreted language and transparent optimizations which can happen without such changes. Next to these two abstract classifications we use two additional implementation-specific properties to separate the optimizations. Optimizations that focus on embedding assumptions or on caching data. By embedding assumptions the low-level VM implementation is made aware of the high-level structures used in the interpreted language. These two groups of classification are not mutually exclusive.

Several non-transparent optimizations which were applied during the evolution of Pinocchio are presented in Chapter 2. The second approach, extensively presented in Chapter 4 are top-down, transparent optimizations. They are compatible with the evaluated language and thus impose no restrictions. Transparent optimization help to separate language and VM design. Both types of optimizations require proper performance estimation using benchmarks. It is important to tackle performance by eliminating the bottlenecks identified by extensive benchmarking and analysis instead of following a trial and error approach. For the optimizations presented in Chapter 4 we show the performance impact by relying on a limited set of three benchmarks. With the in-depth analysis we can show the contribution of each optimization to the overall speedup of an order of magnitude presented in Figure 4.20.

5.1 High-level Languages and Optimizations

High-level languages generally try to delay decisions as much as possible. This is done by introducing indirections for instance using by encapsulating behavior and data in objects rather than applying a set of functions on a memory location. But the flexibility comes at a high price in terms of execution speed. Without extensive optimizations a dynamic language such as Smalltalk is around one to three order of magnitudes slower than a static C program. An example of how flexibility directly affects performance can be seen in the evolution of Pinocchio presented in Chapter 2. Scheme Pinocchio, an ancestor of Pinocchio features a very flexible execution scheme by encapsulating the evaluation behavior directly on the AST nodes. Even though very interesting concepts can be programmed with little effort, the inferior performance led to a more classical approach. In this case the semantics of the language had to be changed completely. AST nodes no longer directly hosted their own behavior but rather the VM had hard-coded behavior for known nodes. This paradigm shift was clearly non-transparent and had a high impact on the language run on the VM requiring it to change part of its semantics. Later on resulting in the current version of Pinocchio other types of optimizations were applied as discussed in Chapter 4. But in general these optimizations were designed in such a way that there is no or only a minimal impact on the language's semantics. Thus we can clearly distinct two types of optimizations based on their impact on the interpreted language:

Non-transparent Optimizations enforce semantic changes on the language running on top of the VM. A possible example could be to restrict certain behavior to work only with VM known types. An example for that is the boolean primitives in Pharo which do not work with customize subclasses.

Transparent Optimizations do *not* enforce semantic changes on the language running on top of the VM. By contrast optimizations happen hidden from the language transparently in the VM. A most prominent example of such an optimization are just-in-time compilers. Even though they optimize code for certain types the semantics naturally will not change at runtime.

Next to the impact of the optimization on the language we classify them upon certain implementation details.

Embedding assumptions: This group of optimizations focuses on making the low-level VM implementation aware of the high-level structures used in the interpreted language. By embedding the assumptions for common use cases into the VM execution speed can be greatly improved. The goal is to reduce the overhead introduced by the semantic mismatch of the VM definition language – generally C – and the interpreted language. A common example is the replication of high-level data structures on the C level.

Caching values focuses on avoiding the repeated calculation of the same values. An example for frequently used caches are inline caches.

These four different categories are not mutually exclusive and the transparency property in particular overlaps with the last two properties.

As mentioned before both of the two optimization types were applied during the evolution of Pinocchio. But the non-transparent optimizations were only used in the early stage of Pinocchio. At this point the design of the language was still under discussion and thus semantic changes were more likely to happen. It is of advantage that we started with system that is more flexible than most common scripting languages available. Sacrificing some flexibility still results in a VM supporting a very flexible language. However if the only concern about a language is speed most certainly the

semantics and syntax of a language will suffer. To assure a good language design we suggest to separate it from the optimization concerns as far as possible. Of course in certain situations it is inevitable to let some part of the optimizations influence the language design. But this should only happen if a significant performance boost, potentially more than an order of magnitude, can be achieved. This requires to clearly order the optimizations by their performance impact. As shown in Chapter 4 not all optimization resulted in the expected performance boost. All these results have been obtained by evaluating several benchmarks. Relying on proper benchmark coverage and evaluation is required to track the performance of a VM and to locate performance bottlenecks. In order to provide useful benchmarks the following three core aspects of benchmarking have to be respected:

Reproducible results: Benchmarks should work similar to unit tests and have to be reproducible with a minimal effort.

Micro benchmarks: Focusing on micro benchmarks helps to identify performance critical sections of a high-level language VMs. This is again similar to unit tests which help to isolate bugs by writing tests for single features.

Statistical sound evaluation: The results from the benchmark runs have to be presented with respect to a basic statistical evaluation.

By using the PBenchmark framework presented in Section 3.2 we provide a set of benchmarks which can be easily reproduced. Additionally the framework assures proper evaluation and presentation of the benchmark results by respecting the statistical background presented in Section 3.1. PBenchmark provides benchmark suites which encourage writing of small micro benchmarks. Furthermore each suite provides a summarized total run time of all the included benchmarks. This way there is no need to write generalized benchmarks which provide only limited meaningful results.

Additionally to the three objective core requirements of benchmarks we can add a fourth suggestion which states that benchmarks should be used in way similar to unit tests. Unit tests focus on testing a small functionality to provide instant feedback on a broken feature, which helps to locate erroneous code. Benchmarks work in a similar fashion. Relying on micro benchmarks which test only part of an optimization provide localized information about a possible performance bottleneck. Hence for developing VMs, unit tests are as important as properly written benchmarks.

5.2 Make it Work, Make it Right, Make it Fast

While working on Pinocchio we spent many hours in figuring out how to optimize a certain performance issue. Generally this was done on pure assumptions with limited scientific support from a benchmark or proper evaluation. As mentioned in the previous section the two main forces behind Pinocchio are speed and flexibility – and generally they do not fit together very well. The temptation to start optimizing part of an immature system are big. For instance Pinocchio did not feature a valid error handling mechanism for several months whilst many man hours were invested in an optimized AST evaluator. The more promising approach is to implement a version which works completely as soon as possible and only then start optimizing. Each optimization should be accompanied with a proper benchmark. The idea behind this is similar to the unit test approach. By writing benchmarks and tracking their progress, the overall influence of an optimization can be better estimated. For instance in cases where there is no speed measurable improvement an optimization can be undone and removed. This has been shown in the case of unwrapped values in Section 4.7 where benchmarks are used to reveal hidden relationships between the different optimizations.

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561. URL http://dx.doi.org/10.1145/73560.73561.

[2] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, Jan. 1984. doi: 10.1145/800017.800542. URL http://dx.doi.org/10.1145/800017.800542.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297033. URL http://dx.doi.org/10.1145/1297105.1297033.

[4] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[5] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802046. URL http://dx.doi.org/10.1145/800055.802046.

[6] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[7] W. R. LaLonde and M. V. Gulik. Building a backtracking facility in Smalltalk without kernel support. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 105–122, Nov. 1988. doi: 10.1145/62083.62094. URL http://dx.doi.org/10.1145/62083.62094.

[8] P. Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0. doi: 10.1145/38765.38821. URL http://dx.doi.org/10.1145/38765.38821.

[9] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5):291–300, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277743. URL http://dx.doi.org/10.1145/277652.277743.

[10] J. W. Simmons, S. Jefferson, and D. P. Friedman. Language extension via first-class interpreters. Technical Report 362, Indiana University Computer Science Department, Sept. 1992. URL http://www.cs.indiana.edu/pub/techreports/TR362.pdf.

[11] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238853. URL http://dx.doi.org/10.1145/1238844.1238853.

[12] T. Verwaest and L. Renggli. Safe reflection through polymorphism. In *CASTA '09: Proceedings of the first international workshop on Context-aware software technology and applications*, pages 21–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-707-3. doi: 10.1145/1595768.1595776. URL http://dx.doi.org/10.1145/1595768.1595776.

Appendices

A.1 Default Benchmarks Sources

A.1.1 Dictionary

benchAtIfAbsentPut

```
1 to: self dictSize * 2 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchAtPut

```
1 to: self dictSize * 2 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchAtPutExisting

```
1 to: self dictSize do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchAtPutNew

```
dict size to: self dictSize * 5 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchDo

```
4 timesRepeat: [
   dict do: [ :i| ]].
```

benchIncludes

```
1 to: self dictSize * 2 by: 73 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchIncludesKey

```
1 to: dict size * 2 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchIncludesKeyExisting

```
1 to: dict size do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchIncludesKeyNew

```
dict size to: dict size * 2 do: [ :i|
   dict at: (self key: i) ifAbsentPut: (self value: i)].
```

benchKeysAndValuesDo

```
4 timesRepeat: [
   dict keysAndValuesDo: [ :k :v| ]].
```

benchRemoveKey

```
1 to: dict size by: 73 do: [ :i|
   dict removeKey: (self key: i) ].
```

A.1.2 Parser

benchAnnotation

```
| string |
string := PEGStringScanner on: 'a
   <abcdefghil>'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

benchBlock

```
| string |
string := PEGStringScanner on: 'a
   ↑ []'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

benchBlockWithArg

```
| string |
string := PEGStringScanner on: 'a
   ↑ [ :a| ]'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

benchBlockWithArgAndBody

```
| string |
string := PEGStringScanner on: 'a
   ↑ [ :a| a]'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

benchSmallMethod

```
| string |
string := PEGStringScanner on: 'a
↑ self'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

benchMediumMethod

```
| string |
string := PEGStringScanner on: 'initialize
internalConstantCode := PDictionary new.
internalConstantCode at: false put: #pushfalse.
internalConstantCode at: true put: #pushtrue.
internalConstantCode at: nil put: #pushnil.
internalConstantCode at: 0 put: #push0.
internalConstantCode at: 1 put: #push1.
internalConstantCode at: 2 put: #push2.'.
```

```
    self repetitionCount timesRepeat: [
       methodParser match: string.
       string reset.].
```

benchLongMethod

```
    | string |
    string := PEGStringScanner on: 'initialize
    |c key bucketIndex values index internalConstantCode
       custom1 custom2 custom3 custom4 custom5 custom6
       custom7 custom8 custom9|
    internalConstantCode := PDictionary new.
    internalConstantCode at: false put: #pushfalse.
    internalConstantCode at: true put: #pushtrue.
    internalConstantCode at: nil put: #pushnil.
    internalConstantCode at: 0 put: #push0.
    internalConstantCode at: 1 put: #push1.
    internalConstantCode at: 2 put: #push2.
    values := Array new: size.
    index := 0.
    self do: [ :value |
       values at: (index := index + 1) put: value ].
    values := Array new: size.
    index := 0.
    self do: [ :value |
       values at: (index := index + 1) put: value ].
    c := 1.
    buckets at: index put: bucket.
    [ c <= bucket bucketSize ] whileTrue: [
          key := bucket at: c.
          bucketIndex := key hash \\ buckets size + 1.
          bucketIndex = index
             ifTrue: [ c := c + 2 ]
             ifFalse: [
                (self bucketWithRoomAt: bucketIndex)
                   newKey: key value: (bucket at: c + 1).
                bucket removeAt: c ] ]
    '.
    self repetitionCount timesRepeat: [
       methodParser parse: string.
       string reset].
```

benchString

```
    | string |
    string := PEGStringScanner on: 'a
    ↑ ''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'''.
    self repetitionCount timesRepeat: [
       methodParser match: string.
       string reset].
```

benchStringNumbers

```
    | string |
    string := PEGStringScanner on: 'a
    ↑ ''012345689'''.
    self repetitionCount timesRepeat: [
```

```
    methodParser match: string.
    string reset].
```

benchSymbol

```
| string |
string := PEGStringScanner on: 'a
↑ #abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'.
self repetitionCount timesRepeat: [
   methodParser match: string.
   string reset].
```

A.1.3 Fibonacci

benchFib

```
31 fib
```